Computational Complexity Theory

Markus Bläser, Holger Dell, Karteek Sreenivasaiah
Universität des Saarlandes

Draft—June 18, 2015 and forever

1 Simple lower bounds and gaps

Complexity theory is the science of classifying problems with respect to their usage of resources. Ideally, we would like to prove statements of the form "There is an algorithm with running time $O(t(n))$ for some problem P and every algorithm solving P has to use at least $\Omega(t(n))$ time". But as we all know, we do not live in an ideal world.

In this chapter, we prove some simple lower bounds. These bounds will be shown for natural problems. (It is quite easy to show lower bounds for "unnatural" problems that are obtained by diagonalization, like in the time and space hierarchy theorems.) Furthermore, these bounds are unconditional. While showing the NP-hardness of some problem can be viewed as a lower bound, this bound relies on the assumption that $\mathsf{P} \neq \mathsf{NP}$.

Unfortunately, the bounds in this chapter will be rather weak. But we do not know any other unconditional lower bounds for "easy" natural problems. While many people believe that $\mathtt{SAT}$ does not have a polynomial time algorithm, even proving a statement like $\mathtt{SAT} \notin \mathsf{DTime}(n^3)$ seems to be out of reach with current methods. We will encounter quite a few such easy looking and "almost" obvious statements that turned out to be really hard and resisted many attacks in the last fifty years.

1.1 A logarithmic space bound

Let $\mathtt{LEN} = \{a^n b^n \mid n \in \mathbb{N}\}$. $\mathtt{LEN}$ is the language of all words that consists of a sequence of as followed by a sequence of b of equal length. This language is one of the examples for a context-free language that is not regular. We will show that $\mathtt{LEN}$ can be decided with logarithmic space and that this amount of space is also necessary. The first part is easy.

Exercise 1.1 *Prove:* $\mathtt{LEN} \in \mathsf{DSpace}(\log)$.

A *small configuration* of a Turing machine M consists of the current state, the content of the work tapes, and the head positions of the work tapes. In contrast to a configuration, we neglect the position of the head on the input tape and the input itself. Since we only consider space bounds, we can assume that M has only one work tape (beside the input tape).

Exercise 1.2 *Let M be an s space bounded 1-tape Turing machine described by $(Q, \Sigma, \Gamma, \delta, q_0, Q_{\text{acc}})$. Prove that the number of small configurations on*

inputs of length m is at most

$$|Q| \cdot |\Gamma|^{s(m)} \cdot (s(m) + 2).$$

If $s = \mathrm{o}(\log)$, then the number of small configurations of M on inputs of length $m = 2n$ is $< n$ for large enough n by the previous exercise.

Assume that there is an s space bounded deterministic 1-tape Turing machine M with $s = \mathrm{o}(\log)$ and $L(M) = \texttt{LEN}$. We consider an input $x = a^p b^n$ with $p \geq n$ and n large enough such that the number of small configurations is $< n$.

Excursus: Onesided versus twosided infinite tapes

In many books, the work tape of a Turing machine is assumed to be twosided infinite. Sometimes proofs get a little easier if we assume that the work tapes are just onesided infinite. The left end of the tape is marked by a special symbol \$ that the Turing machine is not allowed to change and whenever it reads the \$, it has to go to the right. To the right, the work tapes are infinite.

Exercise 1.3 *Show that every Turing machine M with twosided infinite work tapes can be simulated by a Turing machine M' with (the same number of) onesided infinite work tapes. If M is t time and s space bounded, then M' is $\mathrm{O}(t)$ time and $\mathrm{O}(s)$ space bounded. (Hint: "Fold" each tape in the middle and store the two halves on two tracks.)*

Often, the extra input tape is assumed to be twosided infinite. The Turing machine can leave the input and read plenty of blanks written on the tape (but never change them). But we can also prevent the Turing machine from leaving the input on the input tape as follows: Whenever the old Turing machine enters one of the two blanks next to the input, say the one on the lefthand side, the new Turing machine does not move its head. It has a counter on an additional work tape that is increased for every step on the input tape to the left and decreased for every step on the input tape to the right. If the counter ever reaches zero, then the new Turing machine moves its head on the first symbol on the input and goes on as normal. How much space does the counter need? No more than $\mathrm{O}(s(n))$, the space used by the old Turing machine. With such a counter we can count up to $c^{s(n)}$, which is larger than the number of configurations of the old machine. If the old machine would stay for more steps on the blanks of the input tape, then it would be in an infinite loop and the new Turing machine can stop and reject. The time complexity at a first glance goes up by a factor of $s(n)$, since increasing the counter might take this long. There is amortized analysis but the Turing machine might be nasty and always move back and forth between two adjacent cells that causes the counter to be decreased and increased in such a way that the carry affects all positions of the counter. But there are clever redundant counters that avoid this behavior.

We assume in the following that the input tape of M is onesided infinite and the beginning of the input is marked by an extra symbol \$.

An *excursion* of M is a sequence of configurations $C_0 \vdash_M C_1 \vdash_M \ldots \vdash_M C_r$ such that the head of the input tape in C_0 and in C_r are on the \$ or on the first b of x and the head is on an a in all other configurations. An excursion is *small* if in C_0 and C_r, the heads are on the same symbol. It is *large* if the heads are on different symbols.

Lemma 1.1 *Let E be an excursion of M on $x = a^n b^n$ and E' be an excursion of M on $x' = a^{n+n!}b^n$. If the first configuration of E and E' have the same corresponding small configuration and the head of the input tape is on the same symbol, then the last configuration of E and E' have the same corresponding small configuration and the head of the input tape is on the same symbol.*

Proof. If E is small, then E' equals E. Thus, the assertion of the theorem is trivial.

Let E (and E') be large. Assume that the head starts on the \$. The other case is treated symmetrically. Since there are $< n$ small configurations, there must be two positions $1 \leq i < j \leq n$ in the first n symbols of x such that the small configurations S are the same when M first visits the cells in position i and j of the input tape. But then for all positions $i + k(j-i)$, M must be in configuration S as long as the symbol in the cell $i + k(j-i)$ is still an a. In particular, M on input x' is in S when it reaches the cell $i + \frac{n!}{j-i}(j-i) = i + n!$. But between position i and n on x and $i + n!$ and $n + n!$, M will also run through the same small configurations. Thus the small configurations at the end of E and E' are the same. ∎

Theorem 1.2 $\texttt{LEN} \notin \mathsf{DSpace}(o(\log n))$.

Proof. Assume that there is an s space bounded 1-tape Turing machine M for LEN with $s = \mathrm{o}(\log)$. Using Lemma 1.1, we can show by induction that whenever the head on the input tape of M is on the \$ or the first b (and was on an a the step before) then on input $x = a^n b^n$ and $x' = a^{n+n!}b^n$, M is in the same small configuration. If the Turing machine ends its last excursion, then it will only compute on the a's or on the b's until it halts. Since on both inputs x and x', M was in the same small configuration, it will be in the same small configurations until it halts. Thus M either accepts both x and x' or rejects both. In any case, we have a contradiction. ∎

1.2 Quadratic time bound for 1-tape Turing machines

Let $\texttt{COPY} = \{w\#w \mid w \in \{a,b\}^*\}$. We will show that COPY can be decided in quadratic time on deterministic 1-tape Turing machines but not in subquadratic time. Again, the first part is rather easy.

Exercise 1.4 *Show that* $\texttt{COPY} \in \mathsf{DTime}_1(n^2)$. *(Bonus: What about deterministic* 2*-tape Turing machines?)*

Let M be a t time bounded deterministic 1-tape Turing machine for COPY. We will assume that M always halts on the end marker \$.

Exercise 1.5 *Show that the last assumption is not a real restriction.*

Definition 1.3 *A* crossing sequence *of M on input x at position i is the sequence of the states of M when moving its head from cell i to $i+1$ or from cell $i+1$ to i. We denote this sequence by* $\mathrm{CS}(x,i)$

If q is a state in an odd position of the crossing sequence, then M is moving its head from the left to the right, if it is in an even position, it moves from the right to the left.

Lemma 1.4 *Let $x = x_1x_2$ and $y = y_1y_2$. If $\mathrm{CS}(x,|x_1|) = \mathrm{CS}(y,|y_1|)$ then $x_1x_2 \in L(M) \iff x_1y_2 \in L(M)$.*

Proof. Since the crossing sequences are the same, M will behave the same on the x_1 part regardless whether there is x_2 or y_2 standing to the right of it. Since M always halts on \$, the claim follows. ∎

Theorem 1.5 $\texttt{COPY} \notin \mathsf{DTime}_1(\mathrm{o}(n^2))$.

Proof. Let M be a deterministic 1-tape Turing machine for COPY. We consider inputs of the form $x = w\#w$ with $w = w_1w_2$ and $|w_1| = |w_2| = n$. For all $v \neq w_2$, $\mathrm{CS}(x,i) \neq \mathrm{CS}(w_1v\#w_1v,i)$ for all $2n+1 \leq i \leq 3n$ by Lemma 1.4, because otherwise, M would accept $w_1w_2\#w_1v$ for some $v \neq w_2$.

We have $\mathrm{Time}_M(x) = \sum_{i\geq 0} |\,\mathrm{CS}(x,i)|$ where $|\,\mathrm{CS}(x,i)|$ is the length of the sequence. Thus

$$\begin{aligned}\sum_{w_2\in\{a,b\}^n} \mathrm{Time}_M(w_1w_2\#w_1w_2) &\geq \sum_{w_2}\sum_{\nu=2n+1}^{3n} |\,\mathrm{CS}(w_1w_2\#w_1w_2,\nu)| \\ &= \sum_{\nu=2n+1}^{3n}\sum_{w_2} |\,\mathrm{CS}(w_1w_2\#w_1w_2,\nu)|.\end{aligned}$$

All the crossing sequences $\mathrm{CS}(w_1w_2\#w_1w_2,\nu)$ have to be pairwise distinct for all $w_2 \in \{a,b\}^n$. Let ℓ be the average length of such a crossing sequence, i.e., $\sum_{w_2} |\,\mathrm{CS}(w_1w_2\#w_1w_2,\nu)| = 2^n \cdot \ell$.

If ℓ is the average length of a crossing sequence, then at least half of the crossing sequences have length $\leq 2\ell$. There are at most $(|Q|+1)^{2\ell}$ crossing

sequences of length $\leq 2\ell$. Let $c = |Q| + 1$. Then $c^{2\ell} \geq 2^n/2$. Thus $\ell \geq c'n$ for some appropriate constant c'. This yields

$$\sum_{w_2} \mathrm{Time}_M(w_1 w_2 \# w_1 w_2) \geq \sum_{\nu=2n+1}^{3n} 2^n \cdot c'n = c' \cdot 2^n \cdot n^2.$$

For at least one w_2,

$$\mathrm{Time}_M(w_1 w_2 \# w_1 w_2) \geq c' \cdot n^2. \qquad \blacksquare$$

Exercise 1.6 *Show for the complement of* COPY, $\overline{\mathtt{COPY}} \in \mathsf{NTime}_1(n \log n)$.

1.3 A gap for deterministic space complexity

Definition 1.6 *An* extended crossing sequence *of M on input x at position i is the sequence of the small configurations of M when moving its head from cell i to $i+1$ or from cell $i+1$ to i on the input tape. We denote this sequence by* $\mathrm{ECS}(x, i)$.

Theorem 1.7 $\mathsf{DSpace}(\mathrm{o}(\log\log n)) = \mathsf{DSpace}(\mathrm{O}(1))$.

Proof. Assume that there is a Turing machine M with space bound $s(n) := \mathrm{Space}_M(n) \in \mathrm{o}(\log\log n) \setminus \mathrm{O}(1)$. We will show by contradiction that such a machine cannot exists. This proves the theorem.

By Exercise 1.2, the number of small configurations on inputs of length n is $\leq |Q| \cdot |\Gamma|^{s(n)} \cdot (s(n) + 2)$. Since s is unbounded, the number of small configurations can be bounded by $c^{s(n)}$ for large enough n, where c is some constant depending on $|Q|$ and $|\Gamma|$.

In an extended crossing sequence, no small configuration may appear twice in the same direction. Otherwise, a (large) configuration of M would appear twice in the computation of M and M would be in an infinite loop. Thus, there are at most

$$(c^{s(n)} + 1)^{2c^{s(n)}} \leq 2^{2^{ds(n)}}$$

different extended crossing sequences on inputs of length n, where d is some constant. For large enough n_0, $s(n) \leq \frac{d-1}{2} \cdot \log\log n$ for all $n \geq n_0$ and therefore $2^{2^{ds(n)}} < n/2$ for all $n \geq n_0$.

Choose s_0 such that $s_0 > \max\{s(n) \mid 0 \leq n \leq n_0\}$ and such that there is an input x with $\mathrm{Space}_M(x) = s_0$. Such an s_0 exists because s is unbounded.

Now let x be a shortest input with $s_0 = \mathrm{Space}_M(x)$. Since the number of extended crossing sequences is $< n/2$ by the definition of s_0 and n_0, there are three pairwise distinct positions $i < j < k$ such that $\mathrm{ECS}(x, i) = \mathrm{ECS}(x, j) =$

$\mathrm{ECS}(x, k)$. But now we can shorten the input by either glueing the crossing sequences at positions i and j or positions j and k. On at least one of the two new inputs, M will use s_0 space, since any small configuration on x appears in at least one of the shortened strings. But this is a contradiction, since x was a shortest string. ■

Exercise 1.7 *Let* $L = \{\mathrm{bin}(0)\#\,\mathrm{bin}(1)\#\ldots\#\,\mathrm{bin}(n) \mid n \in \mathbb{N}\}$.

1. *Show that* L *is not regular.*
2. *Show that* $L \in \mathsf{DSpace}(\log\log n)$.

2 Robust complexity classes

Complexity classes

Good complexity classes should have two properties:

1. They should characterize important problems.
2. They should be robust under reasonable changes to the model of computation.

To understand the robustness criterion, consider that the class P of polynomial-time computable problems is the same, no matter if we define it using polynomial-time Turing machines, polynomial-time WHILE programs, polynomial-time RAM machines, or polynomial-time C++ programs – we always get the same class of problems.

Definition 2.1

$$\begin{aligned}
\mathsf{L} &= \mathsf{DSpace}(\mathrm{O}(\log n)) \\
\mathsf{NL} &= \mathsf{NSpace}(\mathrm{O}(\log n)) \\
\mathsf{P} &= \bigcup_{i \in \mathbb{N}} \mathsf{DTime}(\mathrm{O}(n^i)) \\
\mathsf{NP} &= \bigcup_{i \in \mathbb{N}} \mathsf{NTime}(\mathrm{O}(n^i)) \\
\mathsf{PSPACE} &= \bigcup_{i \in \mathbb{N}} \mathsf{DSpace}(\mathrm{O}(n^i)) \\
\mathsf{EXP} &= \bigcup_{i \in \mathbb{N}} \mathsf{DTime}(2^{\mathrm{O}(n^i)}) \\
\mathsf{NEXP} &= \bigcup_{i \in \mathbb{N}} \mathsf{NTime}(2^{\mathrm{O}(n^i)}).
\end{aligned}$$

L and NL are classes of problems that can be decided with very little space, which is important in practice when solving problems whose instance do not fit into RAM memory. Note that by Savitch's theorem, $\mathsf{NL} \subseteq \mathsf{DSpace}(\log^2 n)$. We will also see in later lectures that problems in L and also NL have efficient parallel algorithms. P is the class of polynomial-time computable problems

and usually considered to be feasible or tractable. NP characterizes many important combinatorial optimization problems – as we will see, it characterizes all problems whose solutions can be verified in polynomial time. PSPACE is the class of problems that can be decided with polynomial space, which can be considered feasible if the instances are not too large. Note that by Savitch's theorem, there is no point in defining nondeterministic polynomial space. EXP is the smallest deterministic class known to contain NP, and NEXP is the exponential-time analogue of NP.

We have

$$\mathsf{L} \subseteq \mathsf{NL} \subseteq \mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{PSPACE} \subseteq \mathsf{EXP} \subseteq \mathsf{NEXP},$$

which follows from Theorems 25.9 and 25.10 of the "Theoretical Computer Science" lecture. By the time and space hierarchy theorems, we know that $\mathsf{L} \subsetneq \mathsf{PSPACE}$ and $\mathsf{P} \subsetneq \mathsf{EXP}$. Thus, some of the inclusions above must be strict. We conjecture that all of them are strict. By the translation technique, $\mathsf{L} = \mathsf{P}$ would imply $\mathsf{PSPACE} = \mathsf{EXP}$, etc.

2.1 Reduction and completeness

Let R be some set of functions $\Sigma^* \to \Sigma^*$. A language L' is called R *many-one reducible* to another language L if there is some function $f \in R$ (the *reduction*) such that for all $x \in \Sigma^*$,

$$x \in L' \iff f(x) \in L.$$

Thus we can decide membership in L' by deciding membership in L. Let C be some complexity class. A language L is called C-*hard* with respect to R many-one reductions if for all $L' \in \mathsf{C}$, L' is R many-one reducible to L. L is called C-*complete* if in addition, $L \in \mathsf{C}$.

To be useful, reductions should fulfill two properties:

- The reduction should be weaker than the presumably harder one of the two complexity classes we are comparing. Particularly, this means that if L' is R many-one reducible to L and $L \in \mathsf{C}$, then L' should also be in C, that is, C is closed under R many-one reductions.
- The reduction should be transitive. In this case, if L' is R many-one reducible to L and L' is C-hard, then L is also C-hard.

The most popular kind of many-one reductions, *polynomial-time many-one reductions*, was introduced by Richard Karp, who used them to define the concept of NP-hardness. When Steve Cook showed that Satisfiability is NP-complete, he defined *polynomial-time Turing reductions*, which is a more permissive kind of reduction. To define this kind of reduction formally, we introduce the following notion. An *oracle Turing machine* M is a multitape

Turing machine that, in addition to its regular work tape, has a distinguished oracle tape which is read/write. Furthermore, it has two distinguished states, a query state $q_?$ and an answer state $q_!$. Let $f : \Sigma^* \to \Sigma^*$ be some total function. M with oracle f, denoted by M^f, works as follows: As long as M^f is not in $q_?$ it works like a normal Turing machine. As soon as M^f enters $q_?$, the content of the oracle tape is replaced by $f(y)$, where y is the previous content of the oracle tape, and the head of the oracle tape is put on the first symbol of $f(y)$. Then M^f enters $q_!$. The content of the other tapes and the other head positions are not changed. Such an oracle query is counted as only one time step. In other words, M^f may evaluate the function f at unit cost. All the other notions, like acceptance or time complexity, are defined as before. We can also have a language L as an oracle. In this case, we take f to be the characteristic function χ_L of L. For brevity, we will write M^L instead of M^{χ_L}.

Let again $L, L' \subseteq \Sigma^*$. Now L' is *Turing reducible* to L if there is a deterministic oracle Turing machine R such that $L' = L(R^L)$. Basically this means that we can solve L' deterministically if we have oracle access to L. Many-one reductions can be viewed as a special type of Turing reductions where we can only query once at the end of the computation and have to return the same answer as the oracle. To be meaningful, the machine R should be time bounded or space bounded. If, for instance, R is polynomial time bounded, then we speak of polynomial-time Turing reducibility.

Popular reductions

- polynomial-time many-one reductions (denoted by $L' \leq_{\mathrm{P}} L$),
- polynomial-time Turing reductions ($L' \leq_{\mathrm{P}}^{\mathrm{T}} L$),
- logspace many-one reductions ($L' \leq_{\log} L$).

2.2 Co-classes

For a complexity class C, co-C denotes the set of all languages $L \subseteq \Sigma^*$ such that $\bar{L} \in \mathsf{C}$. Deterministic complexity classes are usually closed under complementation. For nondeterministic time complexity classes, it is a big open problem whether a class equals its co-class, in particular, whether $\mathsf{NP} = \text{co-}\mathsf{NP}$. For nondeterministic space complexity classes, this problem is solved.

Theorem 2.2 (Immerman, Szelepcsényi) *For every space constructible* $s(n) \geq \log n$,

$$\mathsf{NSpace}(s) = \text{co-}\mathsf{NSpace}(s).$$

We postpone the prove of this theorem to the next chapter.

Exercise 2.1 *Show that satisfiability is* co-NP*-hard under polynomial time Turing reductions. What about polynomial-time many-one reductions?*

Exercise 2.2 *Show that if L is* C*-hard under R many-one reductions, then $\bar{L}$ is* co-C*-hard under R many-one reductions.*

3 L, NL, and RL

3.1 Logarithmic space reductions

Polynomial-time reductions are not fine-grained enough to analyze the relationship between problems in NL. This is due to the fact that $\mathsf{NL} \subseteq \mathsf{P}$, and so every problem in NL can be solved in polynomial-time. Instead, we often use logarithmic-space many-one reductions for studying L and NL.

Exercise 3.1 *Let f and g be logarithmic space computable functions $\Sigma^* \to \Sigma^*$. Then $f \circ g$ is also logarithmic space computable.*

Recall how we compute a function $\Sigma^* \to \Sigma^*$ in logspace: We use a multi-tape Turing machine with a constant number of tapes, a read-only input tape, and a write-only output tape; crucially, only the work tapes count towards our space requirements.

Corollary 3.1 *$\leq_{\log}$ is a transitive relation, i.e., $L \leq_{\log} L'$ and $L' \leq_{\log} L''$ implies $L \leq_{\log} L''$.*

Proof. Assume that $L \leq_{\log} L'$ and $L' \leq_{\log} L''$. That means that there are logarithmic space computable functions f and g such for all x,

$$x \in L \iff f(x) \in L'$$

and for all y,

$$y \in L' \iff g(y) \in L''.$$

Let $h = g \circ f$. h is logarithmic space computable by Exercise 3.1. We have for all x,

$$x \in L \iff f(x) \in L' \iff g(f(x)) \in L''.$$

Thus h is a many-one reduction from L to L''. ∎

Lemma 3.2 *Let $L \leq_{\log} L'$.*

1. *If $L' \in \mathsf{L}$, then $L \in \mathsf{L}$,*
2. *If $L' \in \mathsf{NL}$, then $L \in \mathsf{NL}$,*
3. *If $L' \in \mathsf{P}$, then $L \in \mathsf{P}$.*

Proof. Let f be a logarithmic space many-one time reduction from L to L'. Let χ_L and $\chi_{L'}$ be the characteristic functions of L and L'. We have $\chi_L = \chi_{L'} \circ f$.

1. It is clear that a language is in L if and only if its characteristic function is logarithmic space computable. By Exercise 3.1, $\chi_{L'} \circ f$ is logarithmic space computable. Thus, $L \in \mathsf{L}$.

2. This follows from a close inspection of the proof of Exercise 3.1. The proof also works if the Turing machine for the "outer" Turing machine is nondeterministic. (The outer Turing machine is the one that gets the input $f(x)$.)

3. Since f is logarithmic space computable, it is also polynomial time computable, since a logarithmically space bounded deterministic Turing machine can make at most polynomially many steps. Thus $\chi_{L'} \circ f$ is polynomial time computable, if $\chi_{L'}$ is polynomial time computable. ■

Corollary 3.3 *1. If L is NL-hard under logarithmic space many-one reductions and $L \in \mathsf{L}$, then $\mathsf{L} = \mathsf{NL}$.*

2. If L is P-hard under logarithmic space many-one reductions and $L \in \mathsf{NL}$, then $\mathsf{NL} = \mathsf{P}$.

Proof. We just show the first statement, the second one follows in a similar fashion: Since L is NL-hard, $L' \leq_{\log} L$ for all $L' \in \mathsf{NL}$. But since $L \in \mathsf{L}$, $L' \in \mathsf{L}$, too, by Lemma 3.2. Since L' was arbitrary, then claim follows. ■

3.2 s-t connectivity

NL is a *syntactic* class. A class C is called syntactic if we can check for a given Turing machine M whether the resources that M uses are bounded as described by C. (This is only an informal concept, not a definition!) While we cannot check whether a Turing machine M is $\mathrm{O}(\log n)$ space bounded, we can make the Turing machine $\mathrm{O}(\log n)$ space bounded by first marking the appropriate number of cells and then simulate M. Whenever M leaves the marked space, we reject.

Exercise 3.2 *Prove that it is not decidable to check whether a Turing machine is $\log n$ space bounded.*

Syntactic classes usually have generic complete problems. For NL, one such problem is

$$\texttt{Gen-NL} = \{e\#x\#1^s \mid e \text{ is an encoding of a nondeterministic Turing machine that accepts } x \text{ in } (\log s)/|e| \text{ space.}\}$$

Exercise 3.3 *Prove that* Gen-NL *is* NL*-complete.*

But there are also natural complete problems for NL. One of the most important ones is CONN, the question whether there is a path from a given node s to another given node t in a directed graph. CONN plays the role for NL that Satisfiability plays for NP.

Definition 3.4 CONN *is the following problem: Given (an encoding of) a directed graph G and two nodes s and t, decide whether there is a path from s to t in G.*

Theorem 3.5 CONN *is* NL*-complete.*

Proof. We have to show two things: CONN $\in$ NL and CONN is NL-hard.

For the first statement, we construct an $O(\log n)$ space bounded Turing machine M for CONN. We may assume that the nodes of the graph are numbered from $1, \ldots, n$. Writing down one node needs space $\log n$. M first writes s on the work tape and initializes a counter with zero. During the whole simulation, the work tape of M will contain one node and the counter. If v is the node currently stored, then M nondeterministically chooses an edge (v, u) of G and replaces v by u and increases the counter by one. If u happens to be t, then M stops and accepts. If the counter reaches the value n, then M rejects.

It is easy to see that if there is a path from s to t, then there is an accepting computation path of C, because if there is a path, then there is one with at most $n-1$ edges. If there is no path from s to t, then C will never accept. (We actually do not need the counter, it is just used to cut off infinite (rejecting) paths.) C only uses $O(\log n)$ space.

For the second statement, let $L \in$ NL. Let M be some $\log n$ space bounded nondeterministic Turing machine for L. We may assume w.l.o.g. that for all inputs of length n, there is one unique accepting configuration.[1] M accepts an input x, if we can reach this accepting configuration from $SC(x)$ in the configuration graph. We only have to consider $c^{\log n} = \text{poly}(n)$ many configurations.

[1] We can assume that the worktape of M is onesided infinite. Whenever M would like to stop, then it erases all the symbols it has written and then moves its head to the \$ that marks the beginning of the tape, moves the head of the input tape one the first symbol, and finally halts.

It remains to show how to compute the reduction in logarithmic space, that is, how to generate the configuration graph in logarithmic space. To do this, we enumerate all configurations that use $s(|x|)$ space where x is the given input. For each such configuration C we construct all possible successor configurations C' and write the edge (C, C') on the output tape. To do so, we only have to store two configurations at a time. Finally, we have to append the starting configurations $\mathsf{SC}(x)$ as s and the unique accepting configuration as t to the output. ■

3.3 Proof of the Immerman–Szelepcsényi theorem

Our goal is to show that the complement of CONN is in NL. Since CONN is NL-complete, $\overline{\texttt{CONN}}$ is co-NL-complete. Thus NL = co-NL. The Immerman–Szelepcsényi theorem follows by translation.

Let $G = (V, E)$ be a directed graph and $s, t \in V$. We want to check whether there is *no* path from s to t. For each d, let

$$N_d = \{x \in V \mid \text{there is a path of length} \leq d \text{ from } s \text{ to } x.\}$$

be the neighbourhood of s of radius d. Let

$$\texttt{DIST} = \{\langle G, s, x, d\rangle \mid \text{there is a path of length} \leq d \text{ from } s \text{ to } x\}$$

DIST is the language of all tuples $\langle G, s, x, d\rangle$ such that x has distance at most d from the source node s. Next we define a partial complement of DIST. A tuple $\langle G, s, x, d, |N_d|\rangle$ is in NEGDIST if there is *no* path from s to x of length $\leq d$. If there is a path from s to x of length $\leq d$, then $\langle G, s, x, d, |N_d|\rangle \notin \texttt{NEGDIST}$. For all $\langle G, s, x, d, S\rangle$ with $S \neq |N_d|$, we do not care whether it is in NEGDIST or not.[2]

Lemma 3.6 DIST $\in$ NL.

Proof. The proof is the same as showing that CONN $\in$ NL. The only difference is that we count to d and not to n. ■

Lemma 3.7 NEGDIST $\in$ NL.

Proof. The following Turing machine accepts NEGDIST:

Input: $\langle G, s, x, d, S\rangle$

1. Guess S pairwise distinct nodes $v \neq x$, one after another.

[2] Strictly speaking, NEGDIST is not one language but a family of languages. When we say that NEGDIST $\in$ NL, we mean that for one choice of the do-not-care triples, the language is in NL.

2. For each v: Check whether v is at a distance of at most d from s by guessing a path of length $\leq d$ from s to v.

3. Whenever one of these tests fails, reject.

4. If all of these tests are passed, then accept

If $S = |N_d|$ and there is no path of length d from s to x, then M has an accepting path, namely the path where M guesses all S nodes v in N_d correctly and guesses the right paths that prove $v \in N_d$. If $S = |N_d|$ and there is a path of length $\leq d$ from s to x, then M can never accept, since there are not $|N_d|$ many nodes different from x with distance $\leq d$ from s.

M is surely $\log n$ space bounded, since it only has to store a constant number of nodes and counters. ∎

If we knew $|N_d|$, then we would be able to decide $\overline{\texttt{CONN}}$ with nondeterministic logarithmic space.

Definition 3.8 *A nondeterministic Turing machine M computes a function $f : \{0,1\}^* \to \{0,1\}^*$ if for every input $x \in \{0,1\}^*$,*

1. *M halts with $f(x)$ on the work tape on every accepting computation path and*

2. *there is at least one accepting computation path on x.*

Note that if $L = L(M)$ for some nondeterministic Turing machine M, then it is not clear whether χ_L is computable (in the sense of definition above) by a nondeterministic Turing machine with the same resources—in contrast to deterministic Turing machines.

Lemma 3.9 *There is a $\log n$ space bounded nondeterministic Turing machine that computes the mapping $\langle G, s, d\rangle \to |N_d|$.*

Proof. We construct a Turing machine M that starts by computing $|N_0|$ and then, given $|N_i|$, it computes $|N_{i+1}|$. Once it has computed $|N_{i+1}|$, it can forget about $|N_i|$.

Input: $\langle G, s, d\rangle$
Output: $|N_d|$

1. Set $|N_0| = 1$.

2. For $i = 0$ to $d - 1$ do

3. $\quad c := 0$

4. $\quad$ For each node $v \in V$ in sequence, nondeterministically guess whether $v \in N_{i+1}$

5. If $v \in N_{i+1}$ was guessed, test whether $\langle G, s, v, i+1 \rangle \in \texttt{DIST}$.
6. If the test fails, reject, else set $c = c + 1$.
7. If $v \notin N_{i+1}$ was guessed, do the following:
8. For all $u \in V$ such that $u \neq v$ and (u, v) is an edge:
9. We run the nondeterministic algorithm in Lemma 3.7 to test $\langle G, s, u, i, |N_i| \rangle \in \texttt{NEGDIST}$
10. If it accepts, we continue the computation
11. If it rejects, we reject
12. $|N_{i+1}| := c$
13. return $|N_d|$

We prove by induction on j that $|N_j|$ is computed correctly.
Induction base: $|N_0|$ is certainly computed correctly.
Induction step: Assume that $|N_j|$ is computed correctly. This means that M on every computation path on which the for loop of line 2 was executed for the value $j-1$ computed the true value $|N_j|$ in line 11. Consider the path on which for each v, M correctly guesses whether $v \in N_{j+1}$. If $v \in N_{j+1}$, then there is a computation path on which M passes the test $\langle G, s, v, j+1 \rangle \in \texttt{DIST}$ in line 5 and increases c in line 6. (Note that this test is again performed nondeterministically.) If $v \notin N_{j+1}$, then for all u such that either $u = v$ or (u, v) is an edge of G, we have that $u \notin N_j$. Hence on some computation path, M will pass all the tests in line 9 and continue the computation in line 10; this is because, by the induction hypothesis, M computed $|N_j|$ correctly, and so the input $\langle G, s, u, i, |N_i| \rangle$ is a valid input for NEGDIST.

On a path on which M made a wrong guess about $v \in N_{j+1}$, M cannot pass the corresponding test and M will reject.

Thus on all paths, on which M does not reject, c has the same value in the end, this value is $|N_{j+1}|$, and there is a least one such path. This proves the claim about the correctness.

M is logarithmically space bounded, since it only has to store a constant number of nodes and the values $|N_j|$ and c. Testing membership in DIST and NEGDIST can also be done in logarithmic space. ∎

Theorem 3.10 $\overline{\texttt{CONN}} \in \mathsf{NL}$.

Proof. $\langle G, s, t \rangle \in \overline{\texttt{CONN}}$ is equivalent to $\langle G, s, t, n, |N_n| \rangle \in \texttt{NEGDIST}$, where n is the number of nodes of G. ∎

Exercise 3.4 (Translation) *Complete the proof of the Immerman–Szelepcsényi Theorem by showing the following: Let s_1, s_2, and f be space constructible such that $n \mapsto s_2(f(n))$ is also space constructible and, for all n, $s_1(n) \geq \log n$, $s_2(n) \geq \log n$, and $f(n) \geq n$. If*

$$DSpace(s_1(n)) \subseteq DSpace(s_2(n)) \,,$$

Then,

$$DSpace(s_1(f(n))) \subseteq DSpace(s_2(f(n))) \,.$$

(Hint: Mimic the proof of the time translation done in class. If the space bounds are sublinear, then we cannot explicitly pad with %s. We do this virtually using a counter counting the added %s.)

3.4 Undirected s-t connectivity

Now that we have found a class that characterizes directed s-t connectivity, it is natural to ask whether we can find a class that describes undirected connectivity. UCONN is the following problem: Given an *undirected* graph G and two nodes s and t, is there a path connecting s and t?

It turns out that undirected connectivity is computationally easier than directed connectivity.

Theorem 3.11 (Reingold) $\texttt{UCONN} \in \mathsf{L}$.

To appreciate this highly non-trivial result, note that, in space $O(\log n)$, we can basically only store the indices of a constant number of vertices. Now take your favourite algorithm for connectivity in undirected graphs and try to implement it with just logarithmic space. The proof of Reingold's theorem is outside the scope of this course, but we will prove that UCONN is in RL, the class of all problems that can be solved in *randomized* logspace.

Excursus: SL-complete problems

Before Reingold's result, many natural problems were known to be equivalent to UCONN under logspace reductions, and the class of these problems was called SL (which we now know to be equal to L). As a consequence of Reingold's result, these problems were shown to be computable in logspace. Here are some examples:

Planarity testing: Is a given graph planar?

Bipartiteness testing: Is a given graph bipartite?

k-disjoint paths testing: Has a given graph k node-disjoint paths from s to t? (This generalizes UCONN.)

There is also a compendium of SL-complete problems.

Carme Àlvarez and Raymond Greenlaw. A compendium of problems complete for symmetric logarithmic space. *Computational Complexity*, 9:73–95, 2000.

Excursus: Vladimir Trifonov

Vladimir Trifonov is (was?) a poor PhD student at University of Texas who showed that $\texttt{UCONN} \in \mathsf{DSpace}(\log n \log\log n)$[3] at the same time when Omer Reingold showed $\texttt{UCONN} \in \mathsf{L}$. This way, a remarkable result became a footnote (or an excursus).

3.4.1 Finding a path using a random walk

To define randomized logspace, we first define randomized Turing machines: They are deterministic Turing machines with an additional tape, the *random tape*. This tape is infinitely long and each cell is initialized with a symbol from Σ (usually $\Sigma = \{0, 1\}$), which is sampled uniformly at random. The random tape is read-only, and the random tape's head can only ever move to the right (it can stand still and it cannot move to the left). Note that the random tape is sampled at random during the execution of the machine. In an equivalent model, we may say that the machine can toss random coins during the execution, and use the outcome in the computation. We say that a randomized Turing machine M is logspace if it halts on every input, and on every input of length n it uses at most $O(\log n)$ cells of its work tapes – neither the random tape, the input tape, nor the output tape count towards the space restriction.

Let M be a randomized Turing machine and $L \subseteq \Sigma^*$ be a language. We say that M *is an* RL*-machine for* L if M is a randomized logspace machine and, for every input $x \in \Sigma^*$, the following two properties hold:

Completeness. If $x \in L$, then $\Pr(M \text{ accepts } x) \geq \frac{1}{2}$.

Soundness. If $x \notin L$, then $\Pr(M \text{ accepts } x) = 0$.

Note that the probability is taken over the internal randomness of M, that is, the contents of the random tape, which is sampled uniformly at random. Also note that the algorithm M is allowed to err only on input instances in the language L. Then we can define RL as follows:

$$\mathsf{RL} = \left\{ L \;\middle|\; \text{there exists an } \mathsf{RL}\text{-machine } M \text{ for } L \right\}.$$

We are now in position to state the theorem for $\texttt{UCONN}$.

Theorem 3.12 $\texttt{UCONN} \in \mathsf{RL}$.

[3] Savitch's Theorem gives $\texttt{UCONN} \in \mathsf{DSpace}(\log^2 n)$ and the best result at that time was $\texttt{UCONN} \in \mathsf{DSpace}(\log^{4/3} n)$ which was achieved by derandomizing a random walk on the graph. We will come to this later ...

The idea of the algorithm is really simple: We perform a *random walk* on the input graph starting at s. If we ever reach t, we know that s and t are in the same connected component. If, on the other hand, t is not reached even after a relatively long time, then we may believe that t is not in the same connected component as s. The analysis of this idea is non-trivial, so let us state the algorithm RandomWalk(G, s, t) first.

1. Let $v := s$.
2. If $v = t$, return "accept".
3. If we were at step 3 more than $O(n^3 \log n)$ times, we halt and reject.
4. Otherwise, set v to be a neighbor of v, chosen uniformly at random, and goto step 2.

Recall that n is the number of vertices of G. Without the explicit counter at step 3, the algorithm may always have some probability to walk in an infinite loop (for example, if there is no path from s to t). Step 3 ensures that the algorithm halts on every input. Note that the soundness of the algorithm is trivial: If s and t are not in the same connected component, RandomWalk does not ever accept.

To prove the completeness of RandomWalk, we define the random walk matrix of G. This is just the adjacency matrix of G, normalized in such a way that every row and every column sums to exactly 1. To simplify things a bit, we assume that G is 3-regular, that is, every vertex has exactly three neighbors. This assumption is without loss of generality since there is a simple self-reduction for `UCONN` whose output consists of 3-regular graphs.

Exercise 3.5 *Prove that there is a logspace many-one reduction from* `UCONN` *to* `UCONN` *such that the output instances of the reduction are always* 3*-regular.*

The random walk matrix M of a 3-regular graph G is defined as follows:

$$M_{u,v} = \begin{cases} \frac{1}{3} & \text{if } \{u, v\} \in E(G)\,, \\ 0 & \text{otherwise.} \end{cases}$$

Note that all row-sums of M are 1 and that M is symmetric. The reason why M is called the random walk matrix is the following. Let $x \in [0, 1]^n$ with $\sum_{v \in V} x_v = 1$. Such a vector x is a *probability distribution* on the vertices of G – it tells us, with which probability we are at a vertex v. For example, in the beginning of RandomWalk(G, s, t), we have $v = s$, and so we are at s with probability exactly one. This is represented by a vector $\mathbf{s} \in [0, 1]^n$ where $\mathbf{s}_s = 1$ and $\mathbf{s}_u = 0$ for all $u \neq s$. The crucial connection between the random walk and the random walk matrix is the following.

Lemma 3.13 *For all $i \in \mathbb{N}$ and $u \in V(G)$, we have*

$$(M^i\mathbf{s})_u = \Pr\Big(\mathit{RandomWalk}(G,s,t) \text{ is at vertex } v = u \text{ after iteration } i\Big)$$

Proof. The proof is by induction on i. Clearly, after iteration $i = 0$, that is, before the first iteration, we have $v = s$ and M^0 is defined to be the identity matrix I, so the lemma holds in this case. Now let $x = M^{i-1}\mathbf{s}$ and assume that x_u is the probability that the algorithm RandomWalk is at u before iteration i. At any vertex, the algorithm walks to one of the three adjacent vertices. This means that Mx characterizes exactly the probability distribution after one further iteration. More formally, let $u \in V(G)$ and consider $(Mx)_u$. Let v_1, v_2, v_3 be the three neighbors of u. Then $(Mx)_u = \frac{1}{3}(x_{v_1} + x_{v_2} + x_{v_3})$ follows from the definition of M. On the other hand, the probability that RandomWalk is at u after iteration i is exactly the probability that we are at one of u's neighbors (which happens with probability $x_{v_1} + x_{v_2} + x_{v_3}$ by the induction hypothesis) and choose to move to u (which happens with probability exactly $\frac{1}{3}$); moreover, these two events are independent since each random step of RandomWalk is independent from any other. Thus this is also equal to $\frac{1}{3}(x_{v_1} + x_{v_2} + x_{v_3})$ and the lemma follows. ∎

To prove the completeness of RandomWalk, all that remains to do is to prove that $(M^i\mathbf{s})_t$ is relatively large for $i = n^3$ as this corresponds exactly to the probability that RandomWalk accepts. *Spectral analysis* from linear algebra is a tool that is suitable in this context. We will quickly review some preliminaries before we finish the proof of the completeness case.

For real symmetric matrices, linear algebra informs us that M has n real eigenvalues $\lambda_1, \ldots, \lambda_n$. In the following exercise, we encourage the reader to review some facts from linear algebra.

Exercise 3.6 *Let G be a non-empty* regular *graph with possible loops, that is, each vertex has the same degree d. Let M be the* random walk matrix *of G, that is, M is the normalized adjacency matrix $\frac{1}{d}A$, so*

$$M_{uv} = \begin{cases} \frac{1}{d} & \text{if } \{u,v\} \in E(G), \\ 0 & \text{otherwise.} \end{cases}$$

This matrix is symmetric and doubly stochastic*, that is, $\sum_j M_{j,i} = \sum_j M_{i,j} = 1$ holds for all i. Let $\|x\|_p = (\sum_i |x_i|^p)^{1/p}$ be the p-norm of x. Let λ_i be the eigenvalues of M, sorted such that $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_n$ holds. Let $x \in \mathbb{R}^n$. Prove:*

1. $\|x\|_2 \leq \|x\|_1 \leq \sqrt{n} \cdot \|x\|_2$.
2. $\|Mx\|_1 = \|x\|_1$ *holds if, for all i, we have $x_i \geq 0$.*

3. $\lambda_1 = 1$. *What is the corresponding eigenvector u so that $Mu = \lambda_1 u$?*
4. *If $x \perp u$, then $\|Mx\|_2 \leq \max\{|\lambda_2|, |\lambda_n|\} \cdot \|x\|_2$.*
5. $\|x\|_\infty \leq \|x\|_2$ *where* $\|x\|_\infty := \max_i |x_i|$.
6. *Bonus 1:* $\|x\|_\infty = \lim_{p\to\infty} \|x\|_p$.
7. *Bonus 2: $\lambda_2 = 1$ holds if and only if G is not connected.*
8. *Bonus 3: $\lambda_i = 0$ holds for all $i \neq 1$ if and only if G is a clique with all loops.*
9. *Bonus 4: $\lambda_n = -1$ holds if and only if G contains a connected component that is bipartite.*

In light of item 4, it is convenient to define $\lambda_2(M)$ as the second-largest eigenvalue of M *in absolute value*, that is, $\lambda_2(M) = \max\{|\lambda_2|, |\lambda_n|\}$. For ease of notation, we again write λ_2 instead of $\lambda_2(M)$. We will use the following theorem without proof:

Theorem 3.14 *Let M be the random walk matrix of a d-regular graph on n vertices. Then $\lambda_2(M) \leq 1 - \frac{1}{d^2 n^2}$.*

Now for the proof of the completeness case, assume that the input graph G is connected. Let y^i be the vector $M^i\mathbf{s}$ for $0 \leq i \leq n^3$. Then we can decompose y^i into a component parallel to the first eigenvector $u = (\frac{1}{n}, \ldots, \frac{1}{n})$ of M and a component orthogonal to it. That is, we have $y^i = u + \epsilon^i$ for some vector ϵ^i with $\epsilon^i \perp u$. Note that $y^i = M^i\mathbf{s} = u + M^i(\mathbf{s} - u)$ and that $(\mathbf{s} - u) \perp u$. Thus we have

$$\begin{aligned}\|M^i(\mathbf{s}-u)\|_\infty &\leq \|M^i(\mathbf{s}-u)\|_2 \\ &\leq \lambda_2^i \cdot \|\mathbf{s}-u\|_2 \leq \lambda_2^i \\ &\leq (1 - \frac{1}{d^2n^2})^i \leq \exp\Big(-i/(3^2n^2)\Big).\end{aligned}$$

This implies $\max_v |\left(M^i(\mathbf{s}-u)\right)_v| \leq \frac{1}{n^2}$ for $i = 18n^2 \log n$, and so $|y^i|_t \geq \frac{1}{n} - \frac{1}{n^2} \geq \frac{1}{2n}$ holds for this choice of i when n is large enough.

What we just proved is that the vertex reached after a random walk with $\Omega(n^2 \log n)$ steps looks very similar to a vertex that is sampled uniformly at random from the connected component in which we would start the random walk. Now all that is left is to see how often we have to repeat this sampling process in order to find the vertex t. If we repeat the process $100n$ times, the probability that we never sample t is $(1 - \frac{1}{n})^{100n} \leq e^{-100}$ in the perfectly uniform case, which is good enough (recall that, by definition of RL, we only need to find t with probability at least $1/2$). In the random walk case, the probability to not find t even though it is in the same connected component

as s is at most $(1 - \frac{1}{2n})^{100n} \leq e^{-50}$, which proves the completeness of the theorem. Note that we "repeat" the $O(n^2 \log n)$-step random walk only in the analysis – the actual random walk will run for $100n \cdot 18n^2 \log n$ steps.

4 Boolean circuits

In this chapter we study another model of computation, Boolean circuits. This model is useful in at least three ways:

- Boolean circuits are a natural model for parallel computation.
- Boolean circuits serve as a nonuniform model of computation. (We will explain what this means later on.)
- Evaluating a Boolean circuit is a natural P-complete problem.

4.1 Boolean functions and circuits

We interpret the value 0 as Boolean false and 1 as Boolean true. A function $\{0,1\}^n \to \{0,1\}^m$ is called a Boolean function. n is its arity, also called the input size, and m is its output size.

A *Boolean circuit* C with n inputs and m outputs is an acyclic digraph with $\geq n$ nodes of indegree zero and m nodes of outdegree zero. Each node has either indegree zero, one or two. If its indegree is zero, then it is labeled with $x_1, \ldots, x_n$ or 0 or 1. Such a node is called an input node. If a node has indegree one, then it is labeled with $\neg$. Such a node computes the Boolean Negation. If a node has indegree two, it is labeled with $\vee$ or $\wedge$ and the node computes the Boolean Or or Boolean And, respectively. The nodes with outdegree zero are ordered. The *depth* of a node v of C is the length of a longest path from a node of indegree zero to v. (The length of a path is the number of edges in it.) The depth of v is denoted by $\mathrm{depth}(v)$. The depth of C is defined as $\mathrm{depth}(C) = \max\{\mathrm{depth}(v) \mid v \text{ is a node of } C\}$. The size of C is the number of nodes in it and is denoted by $\mathrm{size}(C)$.

Such a Boolean circuit C computes a Boolean function $\{0,1\}^n \to \{0,1\}^m$ as follows. Let $\xi \in \{0,1\}^n$ be a given input. With each node, we associate a value $\mathrm{val}(v,\xi) \in \{0,1\}$ computed at it. If v is an input node, then $\mathrm{val}(v,\xi) = \xi_i$, if v is labeled with x_i. If v is labeled with 0 or 1, then $\mathrm{val}(v,\xi)$ is 0 or 1, respectively. This defines the values for all nodes of depth 0. Assume that the value of all nodes of depth d are known. Then we compute $\mathrm{val}(v,\xi)$ of a node v of depth $d+1$ as follows: If v is labeled with $\neg$ and u is the predecessor of v, then $\mathrm{val}(v,\xi) = \neg\,\mathrm{val}(u,\xi)$. If v is labeled with $\vee$ or $\wedge$ and u_1, u_2 are the predecessors of v, then $\mathrm{val}(v,\xi) = \mathrm{val}(u_1,\xi) \vee \mathrm{val}(u_2,\xi)$ or $\mathrm{val}(v,\xi) = \mathrm{val}(u_1,\xi) \wedge \mathrm{val}(u_2,\xi)$. For each node v, this defines a function $\{0,1\}^n \to \{0,1\}$ computed at v by $\xi \mapsto \mathrm{val}(v,\xi)$. Let $g_1, \ldots, g_m$ be the

functions computed at the output nodes (in this order). Then C computes a function $\{0,1\}^n \to \{0,1\}^m$ defined by $\xi \mapsto g_1(\xi)g_2(\xi)\dots g_m(\xi)$. We denote this function by $C(\xi)$.

The labels are taken from $\{\neg, \vee, \wedge\}$. This set is also called *standard basis*. This standard is known to be complete, that is, for any Boolean function $f : \{0,1\}^n \to \{0,1\}^m$, there is Boolean circuit (over the standard basis) that computes it. For instance, the CNF of a function directly defines a circuit for it. (Note that we can simulate one Boolean And or Or of arity n by $n-1$ Boolean And or Or of arity 2.)

Finally, a circuit is called a *Boolean formula* if all nodes have outdegree ≤ 1.

Exercise 4.1 *Show that for any Boolean circuit of depth d, there is an equivalent Boolean formula of depth $\mathrm{O}(d)$ and size $2^{\mathrm{O}(d)}$.*

Exercise 4.2 *Prove that for any Boolean circuit C of size s, there is an equivalent one C' of size $\leq 2s+n$ such that all negations have depth 1 in C'. (Hint: De Morgan's law. It is easier to prove the statement first for formulas.)*

Boolean circuits can be viewed as a model of parallel computation, since a node can compute its value as soon as it knows the value of its predecessor. Thus, the depth of a circuits can be seen as the time taken by the circuit to compute the result. Its size measures the "hardware" needed to built the circuit.

Exercise 4.3 *Every Boolean function $f : \{0,1\}^n \to \{0,1\}$ can be computed by a Boolean circuit of size $2^{\mathrm{O}(n)}$.*[1]

4.2 Uniform families of circuits

There is a fundamental difference between circuits and Turing machines. Turing machines compute functions with variable input length, e.g., $\Sigma^* \to \Sigma^*$. Boolean circuits only compute a function of fixed size $\{0,1\}^n \to \{0,1\}^m$. Since the input alphabet Σ is fixed, we can encode the symbols of Σ by a fixed length binary code. In this way, we overcome the problem that Turing machines and Boolean circuits compute on different symbols. To overcome the problem that circuits compute functions of fixed length, we will introduce families of circuits.

In the following, we will only look at Boolean circuits with one output node, i.e., circuits that decide languages. Most of the concepts and results

[1] This can be sharpened to $(1+\epsilon)\cdot 2^n/n$ for any $\epsilon > 0$. The latter bound is tight: For any $\epsilon > 0$ and any large enough n, there is a Boolean function $f : \{0,1\}^n \to \{0,1\}$ such that every circuit computing f has size $(1-\epsilon)2^n/n$. This is called the *Shannon–Lupanow bound*.

presented in the remainder of this chapter also work for circuits with more output nodes, that is, circuits that compute functions.

Definition 4.1 *1. A sequence $C = C_1, C_2, C_3, \ldots$ of Boolean circuits such that C_i has i inputs is called a family of Boolean circuits.*

2. C is s size bounded and d depth bounded if $\mathrm{size}(C_i) \leq s(i)$ *and* $\mathrm{depth}(C_i) \leq d(i)$ *for all i.*

3. C computes the function $\{0,1\}^* \to \{0,1\}$ given by $x \mapsto C_{|x|}(x)$. Since we can interpret this as a characteristic function, we also say that C decides a language.

Families of Boolean circuits can decide nonrecursive languages, in fact any $L \subseteq \{0,1\}^*$ is decided by a family of Boolean circuits. To exclude such phenomena, we put some restriction on the families.

Definition 4.2 *1. A family of circuits is called s space and t time constructible, if there is an s space bounded and t time bounded deterministic Turing machine that given input 1^n writes down an encoding of C_n that is topologically sorted.*

2. A s size and d depth bounded family of circuits C is called logarithmic space uniform *if it is* $\mathrm{O}(\log s(n))$ *space constructible. It is called* polynomial time uniform*, if it is* $\mathrm{poly}(s)$ *time constructible.*

3. We define

$$\begin{aligned} \mathsf{log\text{-}unif\text{-}DepthSize}(d,s) = \{L \mid & \textit{ there is a } d \textit{ depth and } s \textit{ size bounded} \\ & \textit{logarithmic space uniform family of} \\ & \textit{circuits that decides } L.\} \\ \mathsf{P\text{-}unif\text{-}DepthSize}(d,s) = \{L \mid & \textit{ there is a } d \textit{ depth and } s \textit{ size bounded} \\ & \textit{polynomial time uniform family of} \\ & \textit{circuits that decides } L.\} \end{aligned}$$

Note that logarithmic space uniformity implies polynomial time uniformity. Typically, logarithmic space uniformity seems to be the more appropriate concept.

4.3 Simulating families of circuits by Turing machines

Theorem 4.3 *If $L \subseteq \{0,1\}^*$ is decided by a d depth bounded and s space constructible family of circuits C, then*

$$L \in \mathsf{DSpace}(d+s).$$

Proof. Let ξ be the given input, $|\xi| = n$. To evaluate the Boolean circuit C_n at a $\xi \in \{0,1\}^n$, we have to compute $\mathrm{val}(v, \xi)$ where v is the output node of C. To compute $\mathrm{val}(u, \xi)$ for some u we just have to find the predecessors u_1 and u_2 of u (or just one predecessor in the case of a $\neg$ gate or no predecessor in the case of an input gate). Then we compute recursively $\mathrm{val}(u_1, \xi)$ and $\mathrm{val}(u_2, \xi)$. From these to values, we easily obtain $\mathrm{val}(v, \xi)$.

To do this, we would need a stack of size $d(n)$, the depth of C_n. Each entry of the stack basically consists of two nodes. How much space do we need to write down the nodes? Since each node in a circuit has at most 2 predecessors, the number of nodes of C_n is bounded by $2^{d(n)}$. Thus we need $d(n)$ bits to write down a name of a node. Thus our stack needs $\mathrm{O}(d^2(n))$ many bits altogether. While this is not bad at all, it is more than promised in the theorem.

A second problem is the following: How do we get the predecessors of u? Just constructing the whole circuit would take too much space.

The second problem is easily overcome and we saw the solution to it before: Whenever we want to find out the predecessors of a node u, we simulate the Turing machine M constructing C_n, let it write down the edges one by one, always using the space again. Whenever we see an edge of the form (u', u), we have found a predecessor u' of u.

For the first problem, we again use the trick of recomputing instead of storing data. In the stack, we do not explicitly store the predecessors of a node. Instead we just write down which of the at most two predecessors we are currently evaluating (that means, the first or the second in the representation written by M). In this way, we only have to store a constant amount of information in each stack entry. The total size of the stack is $\mathrm{O}(d(n))$. To find the name of a particular node, we have to compute the names of all the nodes that were pushed on the stack before using M. But we can reuse the space each time.

Altogether, we need the space that is used for simulating M, which is $\mathrm{O}(s)$, and the space needed to store the stack, which is $\mathrm{O}(d)$. This proves the theorem. ∎

Remark 4.4 *If we assume that the family of circuits in the theorem is s size bounded and t time constructible, then the proof above shows that $L \in \mathsf{DTime}(t + \mathrm{poly}(s))$. The proof gets even simpler since we can construct the circuit explicitly and store encodings of the nodes in the stack. The best simulation known regarding time is given in the next exercise.*

Exercise 4.4 *If $L \subseteq \{0,1\}^*$ is decided by a s size bounded and t time constructible family of circuits C, then*

$$L \in \mathsf{DTime}(t + s \log^2 s).$$

4.4 Simulating Turing machines by families of circuits

In the "Theoretical Computer Science" lecture, we gave a size efficient simulation of Turing machines by circuits (Lemma 27.5), which we restate here for arbitrary time functions (but the proof stays the same!).

Theorem 4.5 *Let t be a time constructible function, and let $L \subseteq \{0,1\}^*$ be in $\mathsf{DTime}(t)$. Then there is a $\mathrm{O}(t^2)$ time constructible family of circuits that is $\mathrm{O}(t^2)$ size bounded and decides L.*

Remark 4.6 *If t is computable in $\mathrm{O}(\log t)$ space (this is true for all reasonable functions), then the family is also $\mathrm{O}(\log t)$ space constructible.*

Theorem 4.5 is a size efficient construction. The following result gives a depth efficient construction.

Theorem 4.7 *Let $s \geq \log$ be space constructible and let $L \subseteq \{0,1\}^*$ be in $\mathsf{NSpace}(s)$. Then there is a s space constructible family of circuits that is $\mathrm{O}(s^2)$ depth bounded and decides L.*

Before we give the proof, we need some definitions and facts. For two Boolean matrices $A, B \in \{0,1\}^{n \times n}$, $A \vee B$ denotes the matrix that is obtained by taking the Or of the entries of A and B componentwisely. $A \odot B$ denotes the Boolean matrix product of A and B. The entry in position (i,j) of $A \odot B$ is given by $\bigvee_{k=1}^{n} a_{i,k} \wedge b_{k,j}$. It is defined as the usual matrix product, we just replace addition by Boolean Or and multiplication by Boolean And. The mth Boolean power of a Boolean matrix A is defined as

$$A^{\odot m} = \underbrace{A \odot \cdots \odot A}_{m \text{ times}},$$

with the convention that $A^{\odot 0} = I$, the identity matrix.

Exercise 4.5 *Show the following:*

1. *There is an $\mathrm{O}(\log n)$ depth and $\mathrm{O}(n^3)$ size bounded logarithmic space uniform family of circuits C such that C_n computes the Boolean product of two given Boolean $n \times n$ matrices.*

2. *There is an $\mathrm{O}(\log^2 n)$ depth and $\mathrm{O}(n^3 \log n)$ size bounded uniform family of circuits D such that D_n computes the nth Boolean power of a given Boolean $n \times n$ matrix.*

Note that we here deviate a little from our usual notation. We only allow inputs of sizes $2n^2$ and n^2, respectively, for $n \in \mathbb{N}$. We measure the depths and size as a function in n (though it does not make any difference here).

For a graph G with n nodes, the incidence matrix of G is the Boolean matrix $E = (e_{i,j}) \in \{0,1\}^{n \times n}$ defined by

$$e_{i,j} = \begin{cases} 1 & \text{if there is an edge } (i,j) \text{ in } G, \\ 0 & \text{otherwise.} \end{cases}$$

Exercise 4.6 *Let G and E be as above.*

1. *Show that there is a path from i to j in G of length ℓ iff the entry of $E^{\odot \ell}$ in position (i,j) equals 1.*

2. *Show that there is a path from i to j in G iff the entry in position (i,j) of $(I \vee E)^{\odot n}$ equals 1.*

Proof of Theorem 4.7. Let M be an s space bounded nondeterministic Turing machine with $L(M) = L$. We may assume that M has a unique accepting configuration D. On inputs of length n, M has $c^{s(n)}$ many configurations. The idea is to construct the incidence matrix E of the configuration graph and then compute the $c^{s(n)}$th Boolean power of $I \vee E$. M accepts an input x iff the entry in the position corresponding to the pair $(\mathrm{SC}(x), D)$ in the $c^{s(n)}$th Boolean power of $I \vee E$ is 1.

Once we have constructed the matrix, we can use the circuit of Exercise 4.5. The size of the matrices is $c^{s(n)} \times c^{s(n)}$. A circuit for computing the $c^{s(n)}$the power of it is $\mathrm{O}(\log c^{s(n)}) = \mathrm{O}(s(n))$ space constructible and $\mathrm{O}(\log^2(c^{s(n)})) = \mathrm{O}(s^2(n))$ depth bounded.

Thus the only problem that remains is to construct a circuit that given x outputs a $c^{s(n)} \times c^{s(n)}$ Boolean matrix that is the incidence matrix of the configuration graph of M with input x. This can be done as follows: We enumerate all pairs C, C' of possible configurations. In C, the head on the input tape is standing on some particular symbol, say, x_i. If $C \vdash_M C'$ independent of the value of x_i, then the output gate corresponding to (C, C') is 1. If $C \vdash_M C'$ only if $x_i = 0$, then the output gate corresponding to (C, C') computes $\neg x_i$. If $C \vdash_M C'$ only if $x_i = 1$, then the output gate corresponding to (C, C') computes x_i. Otherwise, it computes 0. Thus the circuit computing the matrix is very simple. It can be constructed in space $\mathrm{O}(s)$ since we only have to store two configurations at a time. ∎

Remark 4.8 *Theorem 4.3 and 4.7 yield an alternative proof of Savitch's theorem.*

4.5 Nick's Class

Circuits are a model of parallel computation. To be really faster than sequential computation, we want to have a exponential speedup for parallel

computations. That means if one wants to study circuits as a model of parallelism, the depth of the circuits should be polylogarithmic. On the other hand, we do not want too much "hardware". Thus the size of the circuits should be polynomial.

Definition 4.9

$$\mathsf{NC}_k = \bigcup_{i\in\mathbb{N}} \mathsf{log\text{-}unif\text{-}DepthSize}(\log^k(n), \mathrm{O}(n^i)) \qquad k = 1, 2, 3, \ldots$$
$$\mathsf{NC} = \bigcup_{k\in\mathbb{N}} \mathsf{NC}_k$$

NC stands for "Nick's Class". Nicholas Pippenger was the first one to study such classes. Steve Cook then chose this name.

Obviously,

$$\mathsf{NC}_1 \subseteq \mathsf{NC}_2 \subseteq \cdots \subseteq \mathsf{NC}.$$

By Theorem 4.3 and 4.7 and Remark 4.4

$$\mathsf{NC}_1 \subseteq \mathsf{L} \subseteq \mathsf{NL} \subseteq \mathsf{NC}_2 \subseteq \cdots \subseteq \mathsf{NC} \subseteq \mathsf{P}.$$

Problems in NL *have efficient parallel algorithms*

The inclusion $\mathsf{NC}_1 \subseteq \mathsf{L}$ suggests that logarithmic space uniformity is too strong for NC_1. There are solutions to this problem but we will not deal with it here.

Excursus: The division breakthrough

Often you find NC_k defined with respect to polynomial time uniformity instead of logarithmic space uniformity. One reason might be that for a long time, we knew that integer division was in polynomial time uniform NC_1 but it was not known whether it was in logarithmic space uniform NC_1. This was finally shown by Andrew Chiu in his Master's thesis. After that, Chiu attended law school.

Paul Beame, Steve Cook, James Hoover. Log depth circuits for division and related problems. *SIAM J. Comput.*, 15:994–1003, 1986.
Andrew Chiu, George I. Davida, Bruce E. Litow. Division in logspace-uniform NC_1. *Informatique Théoretique et Applications* 35:259-275, 2001.

---

5 NL, NC, and P

Lemma 5.1 *Let $L \subseteq \{0,1\}^*$ be* P*-complete under logarithmic space many-one reductions. If $L \in$* NC*, then* NC = P*.*

Proof. Let $L' \in \mathsf{P}$ be arbitrary. Since L is P-hard, $L' \leq_{\log} L$. Since $L \in \mathsf{NC}$, this means that there is a uniform family of circuits C that is $\log^k(n)$ depth and $\text{poly}(n)$ size bounded for some constant k and decides L. We want to use this family to get a uniform family of circuits for L' by using the fact that $L' \leq_{\log} L$.

Since $\mathsf{L} \subseteq \mathsf{NC}$, this is clear in principle, but there are some technical issues we have to deal with. First, we have to compute a function f and not decide a language. Second, a logspace computable function can map strings of the same length to images of different length. We deal with these problems as follows:

- Since f is in particular polynomial time computable, we know that $|f(x)| \leq p(|x|)$ for all x for some polynomial p. Instead of mapping x to $f(x)$, we map x to $0^{|f(x)|}1^{p(|x|)-|f(x)|}f(x)0^{p(|x|)-|f(x)|}$, that is, we pad $f(x)$ with 0's to length $p(|x|)$. In front we place another $p(x)$ bits indicating how long the actual string $f(x)$ is and how many 0's were added. This new function, call it f', is surely logarithmic space computable.

- We modify the family of circuits C such that it can deal with the strings of the form $f'(x)$. We only need a circuit for strings of lengths $2p(n)$. Such a circuit consists of copies of all circuits $C_1, C_2, \ldots, C_{p(n)}$. (This is still polynomial size!) C_i gets the first i bits of the second half of $f'(x)$. The first have of the bits of $f'(x)$ is used to decide which of the $p(|x|)$ circuits computes the desired result. Call this new family C'.

- Since f' is logarithmic space computable, the language

  $$B = \{\langle x, i\rangle \mid \text{the } i\text{th bit of } f'(x) \text{ is } 1\}$$

  is in L. Since $\mathsf{L} \subseteq \mathsf{NC}$, there is a logarithmic space uniform family of circuits that decides B. Using these family, we can get circuits that we can use to feed the corresponding bits of $f'(x)$ into C'.

It is easy but a little lengthy to verify that the new family is still logarithmic space constructible. ∎

Thus a P complete problem is neither likely to have an algorithm that uses few space (even a nondeterministic one) nor to have an efficient parallel algorithm.

5.1 Circuit evaluation

Definition 5.2 *The* circuit value problem `CVAL` *is the following problem: Given (an encoding of) a circuit C with n inputs and one output and an input $x \in \{0,1\}^n$, decide whether $C(x) = 1$.*

Since C can only output two different values, the problem `CVAL` is basically equivalent to evaluating the circuit.

Exercise 5.1 `CVAL` *is* P*-complete.*

Excursus: P-complete problems

If a problem is P-complete under logarithmic space many-one reductions, then this means that it does not have an efficient parallel algorithm by Lemma 5.1, unless you believe that NC = P. Here are some more P-complete problems:

Breadth-depth search: Given a graph G with ordered nodes and two nodes u and v, is u visited before v in a breadth-depth search induced by the vertex ordering?

Maximum flow: Given a directed graph G, a capacity function c on the edges, a source s and a target t and a bound f, is there a feasible flow from s to t of value $\geq f$.

Word problem for context-free grammars: Given a word w and a context-free grammar G, is $w \in L(G)$?

The following book contains and abundance of further problems:

Raymond Greenlaw, James Hoover, Walter L. Ruzzo. *Limits to Parallel Computation:* P*-Completeness Theory*, Oxford University Press, 1995.

There are also some interesting problems of which we do not know whether they are P-hard, for instance:

Integer GCD: Given two n-bit integers, compute their greatest common divisor. (This is a search problem, not a decision problem.)

Perfect Matching: Given a graph G, does it have a perfect matching? (We will see this problem again, when we deal with randomization.)

---

6 The polynomial method

In this chapter, we prove a lower bound for the circuit size of constant depth unbounded fanin circuits for PARITY. The lower bounds even hold in the nonuniform setting.

6.1 Arithmetization

As a first step, we represent Boolean functions $\{0,1\}^n \to \{0,1\}$ as polynomials $p \in \mathbb{R}[X_1, \ldots, X_n]$. A Boolean function is *represented* by a polynomial p if

$$p(x) = f(x) \qquad \text{for all } x \in \{0,1\}^n$$

Above, we embed the Boolean values $\{0,1\}$ into $\mathbb{R}$ by mapping 0 (false) to 0 and 1 (true) to 1.

Since $x^2 = x$ for all $x \in \{0,1\}$, whenever a monomial in the polynomial p contains a factor X_i^j, we can replace it by X_i and the polynomial still represents the same Boolean function. Therefore, we can always assume that the representing polynomial is multilinear. In this case, the representing polynomial is unique.

Exercise 6.1 *Let $f : \{0,1\}^n \to \{0,1\}$ be a Boolean function. Show that the mutlilinear polynomial $p \in \mathbb{C}[x_1, x_2, \ldots, x_n]$ that represents f exactly is indeed unique. That is, if p and p' are two polynomials with $f(x) = p(x) = p'(x)$ for all $x \in \{0,1\}^n$, then p and p' are identical as polynomials over $\mathbb{C}[x_1, \ldots, x_n]$.*

Example 6.1

1. *The Boolean AND of n inputs is represented by the degree n polynomial $X_1 X_2 \cdots X_n$.*
2. *The Boolean NOT is represented by $1 - X$.*
3. *The Boolean OR is represented by $1 - (1 - X_1)(1 - X_2) \cdots (1 - X_n)$ (de Morgan's law).*

6.2 Approximation

One potential way to prove lower bounds is the following:

1. Introduce some complexity measure or potential function.

2. Show that functions computed by devices of type X have low complexity.

3. Show that function Y has high complexity.

A complexity measure that comes to mind is the degree of the representing polynomial. However, since Boolean AND and Boolean OR have representing polynomials of degree n, there is no hope that constant depth unbounded fanin circuits have low degree. However, we can show that Boolean AND and Boolean OR can be *approximated* by low degree polynomials in the following sense: A Boolean function is *randomly approximated with error probability* ϵ by a family of polynomials P if

$$\Pr_{p \in P}[p(x) = f(x)] \geq 1 - \epsilon \qquad \text{for all } x \in \{0,1\}^*.$$

Proof overview: This suggest the following route:

1. Unbounded fanin Boolean AND and Boolean OR can be approximated by low degree polynomials, i.e., degree $\mathrm{O}(\log n)$.

2. Boolean functions that are approximated by unbounded fanin circuits of size s and depth d have degree $\mathrm{O}(\log^{d+1} s)$.

3. `PARITY` cannot be approximated by small degree polynomials.

We start with a technical lemma.

Lemma 6.2 *Let S_0 be a set of size n. Let $\ell = \log n + 2$. Starting with S_0, iteratively construct a tower $S_0 \supseteq S_1 \supseteq S_2 \supseteq \cdots \supseteq S_\ell$ by putting each element of S_{i-1} into S_i with probability $1/2$. Then for all non-empty $T \subseteq S_0$,*

$$\Pr[\textit{there is an } i \textit{ such that } |S_i \cap T| = 1] \geq 1/2$$

Proof. We consider three cases:
Bad case: $|T \cap S_\ell| > 1$. We have

$$\begin{aligned}\Pr[|T \cap S_\ell| > 1] &\leq \Pr[|T \cap S_\ell| \geq 1] \\ &\leq n \cdot 2^{-\ell} \\ &\leq 1/4,\end{aligned}$$

since $|T \cap S_\ell| \geq 1$ means that at least one element survived all ℓ coin flips.
Very good case: $|T| = |T \cap S_0| = 1$.
Good case: $|T \cap S_0| > 1$ and there is an i such that $|T \cap S_i| \leq 1$. We can assume that $|T \cap S_{i-1}| = s > 1$. Then the probability that $T \cap S_i$ has exactly one element is the probability that all but one elements do not survive the

coin flip, which is $s \cdot 2^{-s}$ divided by the probability that after the coin flips $|T \cap S_i| \leq 1$. The latter probability is $(s+1) \cdot 2^{-s}$. Thus the overall probability is $s/(s+1) \geq 2/3$.

With probability 3/4, we are in the very good or good case. If we are in the very good or good case, then with probability $\geq 2/3$, there is an i such that $|T \cap S_i| = 1$. Thus we have success with probability at least $3/4 \cdot 2/3 = 1/2$. ∎

Lemma 6.3 *Let $1 > \epsilon > 0$. There is a family of polynomials P of degree $\mathrm{O}(\log(1/\epsilon) \cdot \log n)$ such that*

$$\Pr_{p \in P}[\bigvee_{i=1}^{n} x_i = p(x)] \geq 1 - \epsilon \qquad \text{for all } x \in \{0,1\}^n.$$

Proof. We construct random sets $S_0, \ldots, S_\ell$ like in Lemma 6.2. Let

$$p_1(x) = \left(1 - \sum_{j \in S_0} x_j\right) \cdots \left(1 - \sum_{j \in S_\ell} x_j\right)$$

If all x_j's are zero, then all the sums in p_1 are zero and $p_1(x) = 1$. Next comes the case where at least one $x_j = 1$. Let $T = \{j \mid x_j \neq 0\}$. By Lemma 6.2, there is an i such that $|S_i \cap T| = 1$ with probability $\geq 1/2$, i.e, the ith factor of p_1 is zero with probability $\geq 1/2$. Now instead of one p_1, we take k independent instances $p_1, \ldots, p_k$ and set $\hat{p} = p_1 \ldots p_k$. If all x_j's are zero then $\hat{p}(x) = 1$. If at least one $x_j = 1$, then at least one $p_k(x) = 0$ with probability $\geq 1 - (1/2)^k \geq 1 - \epsilon$ for $k \geq \log(1/\epsilon)$ and henceforth, $\hat{p}(x) = 0$. Thus $1 - \hat{p}(x) = \bigvee_{i=1}^{n} x_i$ happens with probability $\geq 1 - \epsilon$ for all $x \in \{0,1\}^n$. ∎

Exercise 6.2 *Show that $\bigwedge_{i=1}^{n} x_i$ can be approximated in the same way.*

Lemma 6.4 *Let $f : \{0,1\}^n \to \{0,1\}$ be computed by an s size bounded and d depth bounded circuit of unbounded fanin. Then f can be randomly approximated with error probability ϵ by a family of polynomials of degree $\mathrm{O}(\log^d(s/\epsilon) \cdot \log^d(s))$.*

Proof. We replace every OR or AND gate by a random polynomial from a family with error probability ϵ/s. Each polynomial has degree $\mathrm{O}(\log(s/\epsilon) \cdot \log(s))$, since every gate can have at most s inputs. The degree of the polynomial at the output gate is $\mathrm{O}(\log^d(s/\epsilon) \cdot \log^d(s))$, since the composition of polynomials multiplies the degree bounds.

If all polynomials compute their respective gate correctly, then the polynomial computes the function f correctly. The probability that a single polynomial fails is at most ϵ/s. By a union bound, the overall error probability is thus at most $s\epsilon/s = \epsilon$. ∎

Corollary 6.5 *Let $f : \{0,1\}^n \to \{0,1\}$ be computed by an s size bounded and d depth bounded unbounded fanin circuit. Then there is a polynomial p of degree $\mathrm{O}(\log^d(s/\epsilon) \cdot \log^d(s))$ such that*

$$\Pr_{x \in \{0,1\}^n}[f(x) = p(x)] \geq 1 - \epsilon.$$

Proof. For every x, $\Pr_{p \in P}[f(x) = p(x)] \geq 1 - \epsilon$, where P is the corresponding approximating family. Thus, $\Pr_{x,p}[f(x) = p(x)] \geq 1 - \epsilon$. Thus, there must be at least one p that achieves this probability. ∎

Exercise 6.3 *The statement of Corollary 6.5 is sufficient for our proof. Why does the following argument not work: The zero polynomial and the one polynomial approximate the AND and OR polynomial with error probability $1 - 2^{-n}$. Now take a circuit as in Lemma 6.4 and do the same construction with these polynomials.*

6.3 Parity

So far, we identified the truth value 0 with the natural number 0 and the truth value 1 with the natural number 1. Why this looks natural, in the following it is advantageous to work with the representation -1 for the truth value 1 and 1 for the truth value 0. This representation is also called the *Fourier representation.*

The linear function $1 - 2x$ maps 0 to 1 and 1 to -1. Its inverse function is $\frac{1}{2}(1 - x)$. Thus we can switch between these two representations without changing the degrees of the polynomials in the previous sections.

Using the Fourier representation, the parity function can be written as $\prod_{i=1}^{n} x_i$.

Lemma 6.6 *There is no polynomial p of degree $\leq \sqrt{n}/2$ such that*

$$\Pr_{x \in \{-1,1\}^n}[p(x) = x_1 \cdots x_n] \geq 0.9.$$

Proof. Let p be a polynomial of degree $\leq \sqrt{n}/2$. As seen above, we can assume that p is multilinear. Let $A = \{x \in \{-1,1\}^n \mid p(x) = x_1 \cdots x_n\}$.

Let V be the $\mathbb{R}$-vector space of all functions $A \to \mathbb{R}$. Its dimension is $|A|$.

The set M of all multilinear polynomials of degree $\leq (n + \sqrt{n})/2$ forms a vector space, too. A basis of this vector space are all multilinear monomials

of degree $\leq (n+\sqrt{n})/2$. Thus

$$\begin{aligned}
\dim M &= \sum_{i=0}^{(n+\sqrt{n})/2} \binom{n}{i} \\
&= \sum_{i=0}^{n/2} \binom{n}{i} + \sum_{i=n/2+1}^{n/2+\sqrt{n}/2} \binom{n}{i} \\
&\leq \frac{1}{2} \cdot 2^n + \frac{\sqrt{n}}{2} \binom{n}{n/2} \\
&\leq \frac{1}{2} \cdot 2^n + \frac{\sqrt{n}}{2} \frac{2^n}{\sqrt{\pi n/2}} \qquad \text{(Stirling's formula)} \\
&< 0.9 \cdot 2^n.
\end{aligned}$$

Finally we show that every function in V can be represented by an element of M on A. Then

$$|A| = \dim V \leq \dim M < 0.9 \cdot 2^n.$$

and we are done.

There is a natural isomorphism between the vector space of all functions $\{-1,1\}^n \to \{-1,1\}$ and the vector space of *all* multilinear polynomials, cf. Exercise **??**. Let $\prod_{i\in I} x_i$ be some multilinear monomial of degree $> n/2$. Then

$$\prod_{i\in I} a_i = \prod_{i\in I} a_i \left(\prod_{i\notin I} a_i \right)^2 = p(a) \prod_{i\notin I} a_i$$

for all $a \in A$. Thus for all functions $A \to \mathbb{R}$, the monomials of degree $(n+\sqrt{n})/2$ are sufficient to represent them. This concludes the proof. ∎

Corollary 6.7 *1. Any constant depth d circuit that computes parity on n inputs has size at least $2^{\Omega(\sqrt[2d]{n})}$.*

2. Any polynomial size circuit that computes parity on n inputs has depth at least $\Omega(\log n / \log\log n)$.

Excursus: AC

AC_i is the class of all languages that are recognized by logarithmic space uniform unbounded fanin circuits with depth $O(\log^i n)$ and polynomial size. Let $\mathsf{AC} = \bigcup_{i\in\mathbb{N}} \mathsf{AC}_i$.

If a circuit has polynomial size, then every gate can have at most a polynomial number of inputs. Thus we can simulate every unbounded fanin gate by a binary tree of logarithmic depth. Hence we get

$$\mathsf{AC}_0 \subseteq \mathsf{NC}_1 \subseteq \mathsf{AC}_1 \subseteq \mathsf{NC}_2 \subseteq \ldots$$

and $\mathsf{AC} = \mathsf{NC}$. Corollary 6.7 in particular shows that $\mathsf{AC}_0 \neq \mathsf{NC}_1$. We do not know whether any of the other inclusions is strict.

Exercise 6.4 *Show that* $\texttt{PARITY} \in \mathsf{NC}_1$.

7 P and NP

Recall that we defined NP as $\bigcup_c \mathsf{NTime}(n^c)$, which in turn was defined by using non-deterministic Turing machines.

7.1 The verification characterization of NP

We first discuss a very important characterization of NP, which is arguably more natural than the definition based on non-determinism: Namely, NP contains exactly those decision problems whose yes-instances have "nifty proofs", that is, proofs that are polynomially small and that can be verified in polynomial time.

Definition 7.1 *A deterministic polynomial-time Turing machine M is called a* polynomial-time verifier *for $L \subseteq \Sigma^*$ if there is a polynomial p such that the following two properties hold for all $x \in \Sigma^*$:*

- **Completeness.** *If $x \in L$, then there is a string $c \in \{0,1\}^*$ of length $|c| \leq p(|x|)$ such that M accepts $\langle x, c\rangle$.*
- **Soundness.** *If $x \notin L$, then, for all $c \in \{0,1\}^*$ of length $|c| \leq p(|x|)$, the machine M rejects $\langle x, c\rangle$.*

We denote the language L that M verifies by $V(M)$.

For yes-instances $x \in L$, the string c serves as a *certificate* (or *witness* or *proof*) of the fact that x is in L. The completeness says that, whenever a yes-instance and its certificate are given, the verifier can determine efficiently that the instance is indeed a yes-instance. We do not require an analogous certificate to exist for the no-instances; however, due to the soundness condition, we do require that no purported certificate gets accepted by M even though x is a no-instance. That is, any purported proof for a no-instance $x \notin L$ must contain a mistake, and the verifier will catch this mistake.

Note that the language $V(M)$ verified by M is generally different from the language $L(M)$ of strings accepted by the Turing machine M. In particular, $L(M)$ is the binary relation that contains precisely the pairs (x, c) for which $|c| \leq p(|x|)$ holds and M accepts $\langle x, c\rangle$. The following theorem is proven in the "Theoretical Computer Science" lecture (as Theorem 26.5).

Theorem 7.2 *$L \in$ NP if and only if there is a polynomial-time verifier for L.*

Let M be a polynomial-time verifier for L. As described above, we can view $L(M)$ as a binary relation. We denote this relation by R. Instead of writing $(x, c) \in R$, we may also write $R(x, c) = 1$. Thus $R(x, c) = 1$ holds if and only if $M(\langle x, c\rangle) = 1$. A language L is in NP if and only if there is a polynomial p and a polynomial-time computable relation R such that the following holds for all $x \in \Sigma^*$:

$$x \in L \iff \exists c \in \{0,1\}^{p(|x|)} : R(x, c) = 1. \tag{7.1}$$

The string c is the certificate for $x \in L$; in terms of non-deterministic Turing machines, c models the non-deterministic choices that lead the non-deterministic Turing machine to an accepting computation path.

7.2 NP-complete problems

NP characterizes the complexity of an abundance of relevant problems. The *satisfiability problems* form the most prominent family of NP-complete problems.

Definition 7.3

1. CSAT *is the following problem: Given (the encoding of) a Boolean circuit C, decide whether there is a Boolean vector ξ with $C(\xi) = 1$.*
2. SAT *is the following problem: Given (the encoding of) a Boolean formula in CNF, decide whether it has a satisfying assignment.*
3. ℓSAT *is the following problem: Given (an encoding of) a Boolean formula in ℓ-CNF, decide whether it has a satisfying assignment.*

Because ℓSAT is just a special case of SAT, which in turn is a special case of CSAT, we have

$$\ell\mathtt{SAT} \leq_{\mathrm{P}} \mathtt{SAT} \leq_{\mathrm{P}} \mathtt{CSAT}\,.$$

We previously proved that 3SAT is NP-complete, which implies that all three problems are NP-complete. Note that we usually use polynomial-time many-one reductions to compare problems in NP. However, we do not know of any problem that is NP-complete under polynomial-time many-one reductions but not complete under logarithmic-space polynomial time reductions.

In the following, we write $\exists^P y$ and $\forall^P y$ instead of $\exists y \in \{0,1\}^{p(|x|)}$ and $\forall y \in \{0,1\}^{p(|x|)}$, respectively, for some polynomial p.

7.3 Branching algorithms for SAT

The most straightforward *branching algorithm* for SAT works as follows: Let F be the input CNF formula and let x be a variable of F. Then we know that

the variable is either set to true or it is set to false in a satisfying assignment of F. Hence we can define the two formulas F_0 and F_1, which are obtained from F by setting x either to 0 or to 1: In F_b we set x to $b \in \{0,1\}$, and furthermore, we remove all literals that are equal to 0 and we remove all clauses that are already satisfied because they contain a literal set to 1. Then F is satisfiable if and only if F_0 is satisfiable or F_1 is satisfiable. Note that, in any reasonable encoding, the length of F_0 and F_1 is smaller than the length of F. If we recursively apply this algorithm to F_0 and F_1, we end up with a formula that contains no variables; in this case, either the formula contains the empty clause and is not satisfiable, or the formula is itself empty and thus satisfiable.

We use this opportunity to remark a small improvement to the algorithm above: Whenever we observe a unit clause $\{\ell\}$, we can set the literal ℓ to true instead of branching on the underlying variable. This additional rule is called *unit propagation*. Now the following is an important class of SAT-algorithms: we use unit propagation whenever possible, and if it is not possible, we choose a variable according to some rule and branch on it. Any such algorithm is called a DPLL-algorithm (named after their inventors Davis, Putnam, Logemann, and Loveland). DPLL-algorithms have a degree of freedom: In each branching step, the algorithm gets to choose which variable to branch on next. Practical SAT-solvers are based on the DPLL-paradigm, and they turn out to be extremely efficient on industrial SAT-instances. On the other hand, from a theoretical perspective we cannot explain with current theoretical methods why DPLL-algorithms are so good.

In particular, we can only seem to bound the running time of the DPLL-algorithm by $2^n \cdot \mathrm{poly}(n)$ in the worst case. Surprisingly, perhaps, it is possible to prove a better bound for kSAT: In each branch step, the algorithm chooses the next branch variable uniformly at random from the set of unset variables. This randomized algorithm has an expected running time of at most $2^{\left(1-\frac{1}{k}\right)n} \cdot \mathrm{poly}(n)$ which was proved by Paturi, Pudlák, and Zane (1999). Note that as k tends to ∞ (and the problem kSAT comes closer to SAT), the growth rate of this running time tends to 2. The *strong exponential time hypothesis* (SETH) is that, for every $\epsilon > 0$, there is a k such that kSAT does not have a $(2-\epsilon)^n$-time algorithm. Clearly if SETH is true, then $\mathsf{P} \neq \mathsf{NP}$; the converse is not known and maybe SETH is false.

7.4 Self-reducibility

Search versus Decision

Is showing the existence of a proof easier than finding the proof itself?

Maybe in real life but not for NP-complete problems ...

In practice, we want to *find concrete solutions* to our problems, and it is not usually sufficient merely to know that a solution exists. From the branching algorithm for SAT, we can generalize the concept of "branching" to other decision problems as follows.

Definition 7.4 *A language A is called* downward self-reducible*, if there is a deterministic polynomial-time oracle Turing machine M that satisfies $A = L(M^A)$ and, on input x, the machine M only queries oracle strings of length $< |x|$.*

Without the restriction that M can only query strings of smaller size, this would not be a useful concept since M could query the oracle about the input itself. Using the branching rule from the basic branching algorithm for SAT, we immediately observe the following.

Theorem 7.5 *kSAT, SAT, and CSAT are downward self-reducible.*

Exercise 7.1 *Show that, if A is downward self-reducible, then $A \in$ PSPACE.*

Exercise 7.2 *Let M be a deterministic Turing machine that only queries oracle strings that are shorter than the input string. Show that, if $A = L(M^A)$ and $B = L(M^B)$, then $A = B$.*
Hint: *Prove by induction over n that $A^{\leq n} = B^{\leq n}$ holds.*

For each problem $A \in$ NP, there is a relation R that is polynomial-time computable such that (7.1) holds. But vice versa, each such relation R defines a language in NP via

$$L(R) = \{x \mid \exists^P y : R(x, y) = 1\}.$$

We call R an NP*-relation* or *polynomially bounded relation.* Given an NP-relation R, search(R) is the set of all functions

$$x \mapsto \begin{cases} y & \text{with } R(x, y) = 1 \text{ if such a } y \text{ exists,} \\ \text{undef} & \text{otherwise.} \end{cases}$$

Example 7.6 *Let R be the relation corresponding to* SAT*, i.e., $R(x, y) = 1$ if x encodes a CNF-formula F and y is a satisfying assignment of F.*

1. $L(R) = \texttt{SAT}$.

2. search(R) *is the set of all functions that map a formula* F *to a satisfying assignment if one exists.*

We call search(R) *self-reducible* if there is an $f \in \text{search}(R)$ such that $f = M^{L(R)}$ for some deterministic polynomial-time oracle Turing machine M. Then M is a search-to-decision reduction, that it, it reduces the problem of finding a certificate to the problem of deciding whether there exists a certificate.

Recall that $M^{L(R)}$ denotes that function that is computed by M with oracle $L(R)$. In case $L(R)$ is NP-complete, we will prove below that search(R) is self-reducible. In the example $L(R) = \texttt{SAT}$, we get that, if $\texttt{SAT} \in \mathsf{P}$ holds, then we can find satisfying assignments of satisfiable formulas in polynomial time.

Theorem 7.7 *Let* R *be an* NP*-relation. If* $L(R)$ *is* NP*-complete, then* search(R) *is self-reducible.*

Proof. A deterministic Turing machine M for computing $f \in \text{search}(R)$ works as follows:

Input: x and oracle access to $L(R)$
Output: "undef" or a certificate y such that $R(x, y) = 1$.

1. Ask whether $x \in L(R)$. If no, then output "undef".

2. Let N be some polynomial time verifier that computes R. From N, we get a circuit C such that each satisfying assignment y is a proof that $x \in L(R)$. This construction basically is the same one as the construction that shows that CSAT is NP-complete and can be performed in polynomial time.

3. Since $L(R)$ is NP-complete, there is a polynomial time many one reduction f from CSAT to $L(R)$.

4. Since CSAT is downward selfreducible, we can find a y such that $C(y) = 1$ in polynomial time provided we have an oracle to CSAT. Instead of asking whether $x \in \texttt{CSAT}$, we ask whether $f(x) \in L(R)$. Since f is a many one reduction, these questions are equivalent.

5. Such a y fulfills $R(x, y) = 1$ by construction. Return y.

The above procedure can be performed by a polynomial time deterministic Turing machine with oracle access to $L(R)$. It computes a function in search(R) by construction. Thus search(R) is self-reducible. ∎

In other words, the theorem above says that for NP-complete problems, computing a witness for membership is at most polynomially harder than deciding membership. In practical applications, even just a polynomial overhead may be very disadvantageous; however, many algorithms automatically compute a certificate without additional overhead.

Search problems in the context of NP were introduced by Leonid Levin.

7.5 co-NP

Recall that co-NP is the class of all languages L for which $\bar{L} \in \mathsf{NP}$ holds. Thus L is in co-NP if there is a polynomial time bounded nondeterministic Turing machine M such that, for all $x \in L$, every path in the computation tree of $M(x)$ is accepting, and, for all $x \notin L$, there is at least one rejecting path in the computation tree of $M(x)$. This gives us a characterization in terms of certificates for co-NP: A language L is in co-NP if and only if, for all $x \in \Sigma^*$, we have

$$x \in L \iff \forall y \in \{0,1\}^{p(|x|)} : R(x,y) = 1.$$

Here, R is the polynomial-time computable relation that determines whether the computation path y in the computation tree of $M(x)$ accepts. Hence, co-NP contains precisely those languages whose no-instances can be verified efficiently.

Let UNSAT be the encodings of all unsatisfiable CNF-formulas. Now UNSAT may not be equal to the complement of SAT because the complement of SAT contains all CNF-formulas from UNSAT as well as all strings that are not an encoding of a formula in CNF. However, this is merely a technicality since we can determine in polynomial time whether a given string corresponds to a CNF formula. Thus we always have $\overline{\texttt{SAT}} \leq_\mathrm{P} \texttt{UNSAT} \leq_\mathrm{P} \overline{\texttt{SAT}}$, that is, UNSAT and $\overline{\texttt{SAT}}$ are equivalent under polynomial-time many-one reductions.

A formula F is unsatisfiable if and only if $\neg F$ is a tautology. If F is in CNF, then we can compute the DNF of $\neg F$ in polynomial time by applying De Morgan's law. Hence we have the reduction $\texttt{UNSAT} \leq_\mathrm{P} \texttt{TAUT}$, where TAUT is the following *tautology problem*.

Definition 7.8 TAUT *is given a DNF formula F to decide whether F is a tautology, i.e., whether all assignments satisfy F.*

Note that, if we replaced DNF with CNF here, the resulting problem would be polynomial-time computable. Since SAT is NP-complete, $\overline{\texttt{SAT}}$ is co-NP-complete and so is TAUT.

7.6 NP and co-NP

Since P is closed under complementation, one approach to show that $\mathsf{P} \neq \mathsf{NP}$ would be to show that NP is not closed under complementation, that is, $\mathsf{NP} \neq$

co-NP. On the other hand, to show that NP is closed under complementation, it is sufficient to show that an NP-complete problem is contained in co-NP.

Exercise 7.3 *If* co-NP *contains an* NP*-complete problem, then* NP = co-NP*.*

Most researchers who have an opinion on the matter conjecture that $\mathsf{NP} \neq \mathsf{co\text{-}NP}$. Furthermore, the relationship between P and $\mathsf{NP} \cap \mathsf{co\text{-}NP}$ is open. The following problem is in the intersection of NP and co-NP, but it is not known to be in P. Hence this problem is a candidate for a problem in $(\mathsf{NP} \cap \mathsf{co\text{-}NP}) \setminus \mathsf{P}$.

Definition 7.9 FACTOR *is given two positive integers x and c in binary to decide whether x has a factor b with $2 \leq b \leq c$.*

Exercise 7.4 *Prove the following:*

1. FACTOR $\in$ NP.
2. FACTOR $\in$ co-NP. *(You can use the result by Agrawal, Kayal, and Saxena that* PRIMES $\in$ P*; that is, we can determine in polynomial time whether a given positive integer x is a prime number or not, where the integer x is encoded in binary)*

8 The polynomial time hierarchy

Alternating quantifiers

- Give a theoretical computer scientists some operators!
- Teach her recursion!
- Lean back and watch ...

8.1 Alternating quantifiers

A language L is in NP if and only if there is a polynomial time computable relation R such that the following holds:

$$x \in L \iff \exists^P y : R(x, y) = 1.$$

The string y models the nondeterministic choices of the Turing machine. In the same way, $\mathsf{co\text{-}NP}$ is characterized via

$$x \in L \iff \forall^P y : R(x, y) = 1.$$

Given such a definition as above, theorists do not hesitate to generalize them and see what happens: More precisely, a language is in the class Σ_k^P if there is a polynomial-time computable $(k+1)$-ary relation R such that

$$x \in L \iff \exists^P y_1 \forall^P y_2 \ldots Q^P y_k : R(x, y_1, \ldots, y_{k-1}, y_k) = 1.$$

Above, Q^P stands for $\exists^P$ if k is odd and for $\forall^P$ otherwise. In the same way, the classes Π_k^P are defined: A language is in the class Π_k^P if there is a polynomial-time computable relation R such that

$$x \in L \iff \forall^P y_1 \exists^P y_2 \ldots Q^P y_k : R(x, y_1, \ldots, y_{k-1}, y_k) = 1.$$

By definition, $\mathsf{NP} = \Sigma_1^\mathsf{P}$ and $\mathsf{co\text{-}NP} = \Pi_1^\mathsf{P}$.

The following inclusions are easy to show.

Exercise 8.1 *Show that for all k,*

$$\Sigma_k^\mathsf{P} \subseteq \Sigma_{k+1}^\mathsf{P},$$
$$\Pi_k^\mathsf{P} \subseteq \Pi_{k+1}^\mathsf{P},$$
$$\Sigma_k^\mathsf{P} \subseteq \Pi_{k+1}^\mathsf{P},$$
$$\Pi_k^\mathsf{P} \subseteq \Sigma_{k+1}^\mathsf{P}.$$

The union of these classes

$$\mathsf{PH} = \bigcup_{k \geq 1} \Sigma_k^{\mathsf{P}} = \bigcup_{k \geq 1} \Pi_k^{\mathsf{P}}$$

is called the *polynomial time hierarchy*. We have

$$\mathsf{PH} \subseteq \mathsf{PSPACE},$$

since we can check all possibilities for $y_1, \ldots, y_k$ in polynomial-space.

We can generalize this concept to arbitrary complexity classes. For a polynomial p, let $\exists y, |y| \leq p(n)$ be denoted by $\exists^p x$. For a language L, and a polynomial p,

$$\exists^p L = \{x \mid \exists^p y : \langle x, y \rangle \in L\}.$$

In the same way,

$$\forall^p L = \{x \mid \forall^p y : \langle x, y \rangle \in L\}.$$

For some complexity class C,

$$\exists \mathsf{C} = \bigcup_{p \text{ a polynomial}} \{\exists^p L \mid L \in \mathsf{C}\},$$
$$\forall \mathsf{C} = \bigcup_{p \text{ a polynomial}} \{\forall^p L \mid L \in \mathsf{C}\}.$$

Theorem 8.1 *We have*

$$\Sigma_k^{\mathsf{P}} = \exists \forall \exists \ldots Q \ \mathsf{P},$$
$$\Pi_k^{\mathsf{P}} = \forall \exists \forall \ldots Q \ \mathsf{P}$$

where each sequence consists of k alternating quantifiers.

Proof. The proof is by induction on k.
Induction base: Clearly, $\exists \mathsf{P} = \mathsf{NP} = \Sigma_1^{\mathsf{P}}$ and $\forall \mathsf{P} = \mathsf{co\text{-}NP} = \Pi_1^{\mathsf{P}}$.
Induction step: Let $k > 1$. We only show the induction step for Σ_k^{P}, the case Π_k^{P} is proved in a similar fashion. By the induction hypothesis,

$$\exists \forall \exists \ldots Q \ \mathsf{P} = \exists \Pi_{k-1}^{\mathsf{P}}.$$

Let $L \in \exists \forall \exists \ldots Q \ \mathsf{P}$. This means that there is a language $A \in \Pi_{k-1}^{\mathsf{P}}$ such that $x \in L$ iff there is some y of polynomial length such that $\langle x, y \rangle \in A$. But there is a relation R such that $\langle x, y \rangle$ is in A iff

$$\forall^P y_1 \exists^P y_2 \ldots Q^p y_{k-1} : R(\langle x, y \rangle, y_1, \ldots, y_{k-1}) = 1.$$

Let R' be the relation defined by $R'(x, y, y_1, \ldots, y_{k-1}) = R(\langle x, y \rangle, y_1, \ldots, y_{k-1})$. Clearly R' is polynomial time computable iff R is. Thus $x \in L$ iff

$$\exists^P y \forall^P y_1 \exists^P y_2 \ldots Q^P y_{k-1} : R'(x, y, y_1, \ldots, y_{k-1}) = 1.$$

But this means that $L \in \Sigma_k^{\mathsf{P}}$. Thus $\exists\forall\exists \ldots Q\ \mathsf{P} \subseteq \Sigma_k^{\mathsf{P}}$. Since the argument above can be reversed, the opposite inclusion is true, too. ■

It is an open question whether the polynomial time hierarchy is infinite, that means, $\Sigma_i^{\mathsf{P}} \subsetneq \Sigma_{i+1}^{\mathsf{P}}$ for all i. The next theorem shows that in order to show that the polynomial time hierarchy is not infinite, it suffices to find an i such that $\Sigma_i^{\mathsf{P}} = \Pi_i^{\mathsf{P}}$.

Theorem 8.2 *If $\Sigma_i^{\mathsf{P}} = \Pi_i^{\mathsf{P}}$, then $\Sigma_i^{\mathsf{P}} = \Pi_i^{\mathsf{P}} = \mathsf{PH}$.*

We need the following lemma for the proof.

Lemma 8.3 *For all classes C,*

$$\forall\forall\mathsf{C} = \forall\mathsf{C},$$
$$\exists\exists\mathsf{C} = \exists\mathsf{C},$$

provided that $\langle .,. \rangle$ and the corresponding inverse projections are linear time computable and C is closed under linear time transformations of the input.

Proof. We only prove the first statement, the proof of the second one is similar. Let L be in $\forall\forall\mathsf{C}$. This means that there is a language $A \in \forall\mathsf{C}$ such that

$$x \in L \iff \forall^P y : \langle x, y \rangle \in A$$

Since $A \in \forall\mathsf{C}$, there is some $B \in \mathsf{C}$ such that

$$a \in A \iff \forall^P b : \langle x, y \rangle \in B.$$

Thus

$$x \in L \iff \forall^P y \forall^P b : \langle \langle x, y \rangle, b \rangle \in B.$$

Now we want to replace the two quantifiers by one big one quantifying over $\langle y, b \rangle$. There is only one technical problem: Words in B are of the form $\langle \langle x, y \rangle, b \rangle$ but we need words of the form $\langle x, \langle y, b \rangle \rangle$. Define B' by

$$B' = \{ \langle x, \langle y, b \rangle \rangle \mid \langle \langle x, y \rangle, b \rangle \in B \}. \tag{8.1}$$

B' is again in C, since $\langle \langle x, y \rangle, b \rangle$ is computable in linear time from $\langle x, \langle y, b \rangle \rangle$ and C is closed under linear time transformations of the input.

Now we are almost done but there is one little problem left: We cannot quantify over all $\langle y, b \rangle$, we can only quantify over all $z \in \{0,1\}^{p(n)}$. Some strings z might not correspond to pairs $\langle y, b \rangle$, since either y or b is longer than the polynomial of the corresponding $\forall^P$-quantifier in (8.1). B'' now contains all the words that are in B' and in addition all words $\langle x, z \rangle$ such that z is not a valid pair. Since we can also find out in linear time, whether z is a valid pair, $B'' \in \mathsf{C}$, too.

By construction,

$$x \in L \iff \forall^P z : \langle x, z \rangle \in B''.$$

Thus $L \in \forall C$. ■

Technicalities

The condition of linear time computability and being closed under linear time transformations is fairly arbitrary. Usually, polynomial time computability and being closed under polynomial time reductions is enough. But since $\exists$ and $\forall$ are pretty general operators, we have tried to put as few as possible constraints on C.

(Note that there are linear time computable pairing functions.)

Proof of Theorem 8.2. We show that if $\Sigma_i^\mathsf{P} = \Pi_i^\mathsf{P}$, then $\Sigma_i^\mathsf{P} = \Sigma_{i+1}^\mathsf{P} = \Pi_{i+1}^\mathsf{P}$. Then the theorem follows by induction.

We have $\Sigma_{i+1}^\mathsf{P} = \exists\Pi_i^\mathsf{P}$. Since $\Sigma_i^\mathsf{P} = \Pi_i^\mathsf{P}$, $\Sigma_{i+1}^\mathsf{P} = \exists\Sigma_i^\mathsf{P}$. By Theorem 8.1 and Lemma 8.3, $\Sigma_{i+1}^\mathsf{P} = \Sigma_i^\mathsf{P}$. In the same way we get $\Pi_{i+1}^\mathsf{P} = \Pi_i^\mathsf{P}$. This proves our claim above. ■

Most researchers believe that the polynomial time hierarchy is infinite. So whenever some assumption makes the polynomial time hierarchy collapse, then this assumption is most likely not true (where the probability is taken over the opinions of all researchers in complexity theory). Here "PH collapses" means that $\mathsf{PH} = \Sigma_i$ for some i.

8.2 Complete problems

Let F be a Boolean formula over some variables X. A quantified Boolean formula is a formula of the form

$$Q_1 x_{i_1} \dots Q_n x_{i_n} F(x_1, \dots, x_n).$$

where each Q_i is either $\exists$ or $\forall$. (Note that the x_i are Boolean variables here that can attain values from $\{0, 1\}$ only.) A quantifier alternation is an index j such that $Q_j \neq Q_{j+1}$, i.e., an existential quantifier is followed by a universal one or vice versa. We will always assume that there are no free variables, i.e., the formula is closed.

Definition 8.4 *1.* $\mathtt{QBF}$ *is the following problem: Given a closed quantified Boolean formula, is it true?*

2. $\mathtt{QBF}\Sigma_k$ *is the following problem: Given a closed quantified Boolean formula starting with an existential quantifier and with* $\leq k-1$ *quantifier alternations, is it true?*

3. $\texttt{QBF}\Pi_k$ *is the following problem: Given a closed quantified Boolean formula starting with a universal quantifier and with* $\leq k-1$ *quantifier alternations, is it true?*

We will show in the next chapter that QBF is PSPACE-complete. $\texttt{QBF}\Sigma_k$ and $\texttt{QBF}\Pi_k$ are complete problems for Σ_k^{P} and Π_k^{P}, respectively.

Exercise 8.2 *Show that* $\texttt{QBF}\Sigma_k$ *is* Σ_k^{P}*-complete.*

On the other hand, PH most likely does not have complete problems.

Exercise 8.3 *If* PH *has complete problems, then* PH *collapses.*

8.3 A definition in terms of oracles

For a language A, $\mathsf{DTime}^A(t)$ denotes the set of all languages that are decided by a t time bounded deterministic Turing machine M with oracle A. In the same way, we define $\mathsf{NTime}^A(t)$, $\mathsf{DSpace}^A(s)$, and $\mathsf{NSpace}^A(s)$. If C is a set of languages, then $\mathsf{DTime}^{\mathsf{C}}(t) = \bigcup_{A\in\mathsf{C}} \mathsf{DTime}^A(t)$. In the same way, we define $\mathsf{NTime}^{\mathsf{C}}(t)$, $\mathsf{DSpace}^{\mathsf{C}}(s)$, and $\mathsf{NSpace}^{\mathsf{C}}(s)$. Finally, if T is some set of functions $t : \mathbb{N} \to \mathbb{N}$, then $\mathsf{DTime}^{\mathsf{C}}(T) = \bigcup_{t\in T} \mathsf{DTime}^{\mathsf{C}}(t)$. We do the same for $\mathsf{NTime}^{\mathsf{C}}(T)$, $\mathsf{DSpace}^{\mathsf{C}}(S)$, and $\mathsf{NSpace}^{\mathsf{C}}(S)$.

Let $S_1 = \mathsf{NP}$ and $S_i = \mathsf{NP}^{S_{i-1}}$. In other words, $S_2 = \mathsf{NP}^{\mathsf{NP}}$, $S_3 = \mathsf{NP}^{\mathsf{NP}^{\mathsf{NP}}}$, and so on. S_i is another definition of the polynomial time hierarchy. More precisely, we have the following theorem, where $P_i = \mathsf{co\text{-}NP}^{S_{i-1}}$.

Theorem 8.5 *For all* i, $\Sigma_i^{\mathsf{P}} = S_i$ *and* $\Pi_i^{\mathsf{P}} = P_i$.

Proof. The proof is by induction on i.
Induction base: For $i = 1$, we have $\Sigma_1^{\mathsf{P}} = \mathsf{NP} = S_1$ and $\Pi_1^{\mathsf{P}} = \mathsf{co\text{-}NP} = P_1$.
Induction step: Assume that the claim is valid for i. We first show that $\Sigma_{i+1}^{\mathsf{P}} \subseteq S_{i+1}$. $L \in \Sigma_{i+1}^{\mathsf{P}}$ if there is a polynomial time computable relation R such that

$$x \in L \iff \exists^P y_1 \forall^P y_2 \ldots Q^P y_{i+1} : R(x, y_1, \ldots, y_{i+1}) = 1.$$

Let $L' = \{\langle x, y\rangle \mid \forall^P y_2 \ldots Q^P y_{i+1} : R(x, y, y_2 \ldots, y_{i+1})\}$. By construction, $L' \in \Pi_i^{\mathsf{P}}$. By the induction hypothesis, $L' \in P_i$. The following Turing machine tests whether $x \in L$:

Input: x

1. Guess a y.
2. Query the oracle to check whether $\langle x, y\rangle \in L'$.
3. Accept if the answer is yes, otherwise reject.

This shows that $L \in \mathsf{NP}^{P_i}$. But $S_i = \mathsf{co}\text{-}P_i$. Thus $\mathsf{NP}^{S_i} = \mathsf{NP}^{P_i}$ and we have $L \in \mathsf{NP}^{S_i} = S_{i+1}$.

To show $S_{i+1} \subseteq \Sigma^{\mathsf{P}}_{i+1}$, let $L \in S_{i+1}$. Let M be a polynomial time nondeterministic Turing machine M that decides L with an oracle $W \in S_i$. Let M be t time bounded. W.l.o.g. we may assume that in each step M has at most two nondeterministic choices. Consider M on input x: Let $y \in \{0,1\}^{t(|x|)}$ be a string that describes the nondeterministic choices of M along some computation path. On such a path, M might query W at most $t(|x|)$ times. Let $a \in \{0,1\}^{t(|x|)}$ describe the answers to the queries.[1]

Given y and a, the following function f is polynomial time computable: $f(x,y,a,i)$ is the ith string that M on input x asks the oracle on the path given by y provided that the answers to previous oracle queries are as given by a. $f(x,y,a,i)$ is undefined (represented by some special symbol) if the oracle is asked fewer than i times. To compute $f(x,y,a,i)$, we just simulate M and make the nondeterministic choices as given by y and instead of really asking W, we pretend that the answer is as given by a. With the same simulation, we can also compute the following relation R given by $R(x,y,a) = 1$ iff M accepts x on the path given by y with oracle answers a. Now we have

$$x \in L \iff \exists^P \langle y,a\rangle [R(x,y,a) = 1 \wedge \bigwedge_{j:a_j=1} f(x,y,a,j) \in W \wedge \bigwedge_{j:a_j=0} f(x,y,a,j) \notin W]$$

where an expression involving an undefined $f(a,y,j)$ is always true. Note that the first part of the expression tests whether M has an accepting path and the second part verifies whether a contains the correct answers.

Each oracle answer check is either in $S_i = \Sigma^{\mathsf{P}}_i$ (positive answers) or in $P_i = \Pi^{\mathsf{P}}_i$ (negative answers). Thus we can replace each "$f(x,y,a,j) \in W$" and "$f(x,y,a,j) \notin W$" by a quantified expression with i quantifiers by the induction hypothesis. We can write all the quantifiers in front and combine quantifiers in such a way that we get a quantified expression for L with $i+1$ quantifiers starting with an existential quantifier. Thus $L \in \Sigma^{\mathsf{P}}_{i+1}$.

$\Pi^{\mathsf{P}}_{i+1} = P_{i+1}$ follow from the fact that both classes are the co-classes of $\Sigma^{\mathsf{P}}_{i+1}$ and S_{i+1}, respectively. ∎

[1] If M asks the oracle τ times, then the first τ bits of a are the answer.

9 P, NP, and PSPACE

The main result of this chapter to show that QBF is PSPACE-complete.

Exercise 9.1 *If L is* PSPACE*-hard under polynomial time many one reductions and $L \in$ NP, then* NP = PSPACE*. If $L \in$ P, then* P = PSPACE*.*

Theorem 9.1 QBF *is* PSPACE*-complete*

Proof. We first show that QBF is in PSPACE. We devise a recursive procedure: If F is a quantified Boolean formula without any quantifiers, we can evaluate it in polynomial space using the same procedure that we used to evaluate Boolean circuits. The problem is even easier, since there are no variables in the formula but only constants. Let $F = Qx_iF'$. We replace in F' every free occurrence of x_i by 0 and by 1, respectively. Let F'_0 and F'_1 be the resulting formulas. We now recursively evaluate F'_0 and F'_1. If Q is an $\exists$-quantifier, then F is true if and only if F'_0 is true or F'_1 is true. If Q is a $\forall$-quantifier, then F is true if F'_0 and F'_1 are true.

To implement this procedure, we need a stack whose size is linear in the number of quantifiers. Thus the total space requirement is surely polynomially bounded.

Next we show that QBF is PSPACE-hard. Let L be some language in PSPACE and let M be some deterministic polynomial-space Turing machine with $L(M) = L$. Without loss of generality, we may assume that M has only one work tape and no separate input tape. Furthermore, we may assume that M has a unique accepting configuration. We will encode configurations by bit strings. We encode the state by some fixed length binary representation and also the position of the head. (For the head position, the length is fixed for inputs of the same length.) Each symbol is represented by a binary string of fixed size, too. The configuration is encoded by concatenating all the bit strings above. Since the sizes of the concatenated strings are fixed, this encoding is an injective function. Let $p(n)$ be the length of the encoding on inputs of length n.

Let x be an input of length n. We will construct a formula F_x that is true if and only if $x \in L$. Let s_x denote the encoding of the start and t the encoding of the accepting configuration. Note that M can make at most $2^{p(n)}$ many steps for some polynomial p.

We will inductively construct a formula $F_j(X, Y)$. Here X and Y are disjoint sets of $p(n)$ distinct variables each. Let ξ and η be two encodings of configurations. $F_j(\xi, \eta)$ denotes the formula where we assign each variable

in X a bit from ξ and each variable in Y a bit from η. (Assume that the variables in X and Y are ordered.) We will construct F_j in such a way that $F(\xi, \eta)$ is true if and only if η can be reached from ξ by M with $\leq 2^j$ steps.

The inductions start is easy: $F_0(X, Y) = (X = Y) \vee S(X, Y)$. Here $X = Y$ denotes the formula that compares X and Y bit by bit and is true if and only if all bits are the same. $S(X, Y)$ is true if Y can be reached from X in one step (see the exercise below). The size of F_0 is polynomial in n.

For the induction step, the first thing that comes to mind is to mimic the proof of Savitch's theorem. We try

$$F_j(X, Y) = \exists Z : F_{j-1}(X, Z) \wedge F_{j-1}(Z, X).$$

("$\exists Z$" means that every Boolean variable in Z is quantified with an existential quantifier.) While this formula precisely describes what we want, its size is too big. It is easy to see that the size grows exponentially. Therefore, we exploit the following trick and use the formula F_j "twice":

$$F_j(X, Y) = \exists C \forall A \forall B : F_{j-1}(A, B) \vee (\neg(A = X \wedge B = C) \wedge \neg(A = C \wedge B = Y)).$$

Here $F_{j-1}(A, B)$ is used two times, namely if $A = X$ and $B = C$ or $A = C$ and $B = Y$. Therefore it checks whether X is reachable from Y within 2^j steps. However, its size now is only polynomial, since when going from F_{j-1} to F_j, we only get an additional additive increase of the formula size that is polynomial.

The final formula now is $Q_x = F_{p(n)}(s_x, t)$. By construction, Q_x is true if and only if $x \in L$. ∎

Exercise 9.2 *Construct a formula S such that $S(\xi, \eta)$ is true if and only if $\xi \vdash_M \eta$. Show that S has polynomial size and can be computed in polynomial time.*

Excursus: Games

Many games are PSPACE-hard, more precisely, given a board configuration, deciding whether this is a winning position for one player is PSPACE-hard (but not necessarily in PSPACE, since many games can go on for more than a polynomial number of steps).

To talk about complexity, we have to generalize the games to arbitrarily large boards. For Checkers or the ancient game Go, this is no problem. For Chess, one has to be a little creative: On a board of size $7n + 1$, one has e.g. 1 king, n queens, $2n$ bishops, $2n$ knights, $2n$ rooks, and $7n + 1$ pawns in each color.

While it looks at a first glance astonishing that many games are PSPACE-hard, it is rather natural. Being in a winning position means that for all moves of my opponent I have an answer such that for all moves of my opponent I have an answer ... which is a sequence of alternating quantifiers like in QBF.

Geography: Given a directed graph G with a start node s, two players construct a path by adding an edge to the front of the path until one player cannot add an edge anymore since this would reach a node already visited. Decide whether the first player has a winning strategy on G.

Checkers: Given a board position in a Checkers game, is this a winning position for white?

Go: Given a board position in a Go game, is this a winning position for white?

Chess: Given a board position in a (generalized) Chess game, is this a winning position for white?

All of the games are PSPACE-hard, Geography and some variants of Go are also in PSPACE.

In a symmetric game (i.e, both players can make the same moves and start in the same configuration) where a player can legally "pass" and in the case of a tie, the first player is declared to be the winner, the first player will always win. If the second player had a winning strategy, the first player could steal it by passing. Thus the board positions used for the hardness results above have to be rather exotic.

10 The Karp–Lipton Theorem

The class P is precisely the class of all languages that are decided by polynomial size uniform families of circuits. What happens if we drop the uniformity constraint? That is, we look at families of polynomial size circuits but do not care how to construct them. For reasons that will become clear later, call the resulting class P/poly. Is it then possible that NP $\subseteq$ P/poly? While not as strong as P = NP, this inclusion would have a huge impact: One (governments are a clear candidate) just takes a lot of resources and tries to find the polynomial size circuit for SAT with, say, 10000 variables. This would break any current crypto-system, for instance.

10.1 P/poly

We defined the class P/poly in terms of circuits. One can also define this class in terms of Turing machines that take advice.[1] Such a Turing machine has an additional read-only advice tape. On this tape, it gets an additional advice string, that only may depend on the length of the input.

Definition 10.1 *Let t and a be two functions $\mathbb{N} \to \mathbb{N}$. A language L is in the class $\mathsf{DTime}(t)/a$ if there is a deterministic Turing machine M with running time t and with an additional advice tape and a sequence of strings $\alpha(n) \in \{0,1\}^{a(n)}$ such that the following holds: For all $x \in L$, M accepts x with $\alpha(|x|)$ written on the advice tape. For all $x \notin L$, M rejects x with $\alpha(|x|)$ written on the advice tape. (The advice string is not counted as part of the input!)*

This definition extends to nondeterministic classes and space classes in the obvious way. We can also extend the definition to sets of functions T and A. We define $\mathsf{DTime}(T)/A = \bigcup_{t \in T, a \in A} \mathsf{DTime}(t)/a$. If we choose T and A both to be the class of all polynomials, then we get exactly P/poly.

Lemma 10.2 *For all languages $L \in \{0,1\}^*$, there is a polynomial time Turing machine M with polynomial advice accepting L iff $L \in$ P/poly.*

Proof. Let L be decided by a polynomial time Turing machine M with polynomial advice. Let x be an input of length n and let $\alpha(n)$ be the corresponding advice string. Consider $\alpha(n)$ as a part of the input of M. There is a polynomial size circuit C_n such that $C_n(x, \alpha(n)) = 1$ iff M with

[1] Sometimes, Turing machines are smarter than humans.

input x and advice $\alpha(n)$ accepts and this holds for all x of length n. Now choose C'_n to be the circuit that is obtained from C_n by considering the x as the input and treating the bits of $\alpha(n)$ as constants. C'_n is the desired circuit.

If L is decided by a nonuniform family of polynomial size circuits C_i, then the nth advice string $\alpha(n)$ is just an encoding of C_n. Circuit evaluation can be done in polynomial time. Thus, given an input x of length n, we just have to evaluate C_n at x. ■

For each input length n, we give the Turing machine an advice $\alpha(n)$. Note that we do no restrict this sequence, except for the length. In particular, the sequence need not be to computable at all.

Lemma 10.3 *Any tally language $L \subseteq \{1\}^*$ is in $\mathsf{P/poly}$.*

Proof. For each input length n, we have an advice string of length one. This string is 1 if $1^n \in L$ and 0 otherwise. ■

There are tally sets that are not computable, for instance

$$\{1^n \mid \text{the } n\text{th Turing machine halts on the empy word}\}.$$

10.2 The Karp–Lipton Theorem

The Karp–Lipton Theorem states that $\mathsf{NP} \subseteq \mathsf{P/poly}$ is not very likely, more precisely, this inclusion collapses the polynomial time hierarchy to the second level.

If $\mathsf{NP} \subseteq \mathsf{P/poly}$, then of course $\mathtt{SAT} \in \mathsf{P/poly}$. That is, there is a family C_i of polynomial size circuits for $\mathtt{SAT}$. C_i is a circuit that decides whether a given formula of length *exactly* i is satisfiable or not. But we can also assume that there is a family D_i of polynomial size circuits such that D_i decides whether a given formula of size *at most* i is satisfiable. D_i basically is "the union" of $C_1, \ldots, C_i$. Its size is again polynomial.

Lemma 10.4 *Let D_i be a family as described above. Then there is a polynomial time computable function h such that $h(F, D_{|F|})$ is a satisfying assignment for F iff F is a satisfiable formula.*

Proof. We use the downward self-reducibility for $\mathtt{SAT}$. Choose a variable in F and set it to zero and to one, respectively. Let F_0 and F_1 be the corresponding formulas. Their length is at most the length of F. Now evaluate $D_{|F|}$ to check whether F_0 or F_1 is satisfiable. If one of them is, say F_0, then we can construct a satisfying assignment for F by setting the chosen variable to zero and proceed recursively with F_0. ■

Theorem 10.5 (Karp–Lipton) *If $\mathsf{NP} \subseteq \mathsf{P/poly}$, then $\Pi_2^{\mathsf{P}} \subseteq \Sigma_2^{\mathsf{P}}$.*

Proof. Let $A \in \Pi_2^{\mathsf{P}} = \forall\exists\mathsf{P} = \forall\mathsf{NP}$. Then there is a language $B \in \mathsf{NP}$ such that for all x,

$$x \in A \iff \forall^P y : \langle x, y\rangle \in B.$$

Since $B \in \mathsf{NP}$, there is a polynomial time many one reduction f from B to $\mathtt{SAT}$. In other words, for all x,

$$x \in B \iff f(x) \in \mathtt{SAT}.$$

Thus for all x,

$$x \in A \iff \forall^P y : f(\langle x, y\rangle) \in \mathtt{SAT}.$$

$f(z)$ is polynomially bounded for all z. Since y is polynomially bounded, too, $f(\langle x, y\rangle)$ is polynomially bounded in $|x|$.

Let D_i be a sequence of polynomial size circuits for $\mathtt{SAT}$ as constructed above. By Lemma 10.4, for all x,

$$x \in A \iff \forall^P y : h(f(\langle x, y\rangle), D_{|f(\langle x,y\rangle)|}) \text{ satisfies } f(\langle x, y\rangle),$$

where h is the function constructed in Lemma 10.4.

Note that we only know that the family D_i exists. It could be very hard to construct it. But we can use the power of the $\exists$ quantifier and guess the correct circuit! Since $f(\langle x, y\rangle)$ is polynomially bounded in $|x|$, the size of $D_{|f(\langle x,y\rangle)|}$ is bounded by $p(|x|)$ for some appropriate polynomial. Thus, for all x,

$$\begin{aligned} x \in A \iff \exists^P D : D \text{ is a circuit with } |f(\langle x, y\rangle)| \text{ inputs} \\ \forall^P y : h(f(\langle x, y\rangle), D) \text{ satisfies } f(\langle x, y\rangle). \end{aligned}$$

Note that we implicitly check whether D was the right guess since we verify that h produced a satisfying assignment. Even if this assignment is produced with a "wrong" circuit, we are still happy. Hence, $A \in \Sigma_2^{\mathsf{P}}$. ∎

Note that already $\Pi_2^{\mathsf{P}} \subseteq \Sigma_2^{\mathsf{P}}$ collapses the polynomial time hierarchy to the second level. We only showed it under the condition that $\Pi_2^{\mathsf{P}} = \Sigma_2^{\mathsf{P}}$. But $\Pi_2^{\mathsf{P}} \subseteq \Sigma_2^{\mathsf{P}}$ is sufficient to show that $\Sigma_2^{\mathsf{P}} = \Sigma_3^{\mathsf{P}}$ (apply an $\exists$). Then also $\Pi_2^{\mathsf{P}} = \Pi_3^{\mathsf{P}}$, since these are co-classes. From this, we get the collapse.

11 Relativization

In this chapter, we have a relativized look at the P versus NP question and related questions. More precisely, we will investigate the classes P^A and NP^A for some oracle A. It turns out, that there are oracles for which these two classes coincide and there are oracles for which these two classes are different. While this does not say much about the original question, it shows an important property that a proof technique should possess if it is capable to resolve the P versus NP question. This technique should not "relativize", i.e., it should not be able to resolve the question P^A versus NP^A for arbitrary A. The usual simulation techniques and diagonalization relativizes, in fact, almost all of the techniques from recursion theory relativize.

11.1 P and NP

Theorem 11.1 $\mathsf{P}^A = \mathsf{NP}^A$ *for all* PSPACE*-complete* A.

Proof. Let A be PSPACE-complete. Let L be in NP^A. Let M be a nondeterministic polynomially time bounded Turing machine with $L(M^A) = L$. M can ask only a polynomial number of questions to A on every branch. Each of them is only polynomially long. Thus $L \in \mathsf{PSPACE}^A$. But $\mathsf{PSPACE}^A = \mathsf{PSPACE}$, since instead of querying the oracle A, we can just simulate a polynomially space bounded Turing machine for A.

Since A is PSPACE-hard, there is a many one reduction from any language in PSPACE to A. In particular, $\mathsf{PSPACE} \subseteq \mathsf{P}^A$, since a many one reduction is also a Turing reduction. Putting everything together, we get

$$\mathsf{P}^A \subseteq \mathsf{NP}^A \subseteq \mathsf{PSPACE}^A = \mathsf{PSPACE} \subseteq \mathsf{P}^A. \qquad \blacksquare$$

For a language $B \in \{0,1\}^*$, let $\text{tally}(B) = \{\#^n \mid \text{there is a word of length } n \text{ in } B\}$. A nondeterministic Turing machine M with oracle B can easily decide $\text{tally}(B)$. On input $\#^n$, it just guesses a string $x \in \{0,1\}^n$ and verifies whether $x \in B$ by asking the oracle. If yes, it accepts; otherwise, it rejects.

More precisely: When M reads the input string, it can directly guess the string x on the oracle tape. This takes n steps. When it reads the first blank on the input tape, it enters the query state and gets the answer (one step). Then it just has to check whether the answer on the input tape is 0 or 1 (one step). Thus $\text{tally}(B) \in \mathsf{NTime}^B(n+2) \subseteq \mathsf{NP}^B$.

Lemma 11.2 *There is a* $B \in \mathsf{EXP}$ *such that* $\text{tally}(B) \notin DTime^B(2^n)$.

Proof. In a similar way like we have encoded Turing machines as binary strings, we can also encode oracle Turing machines. The encoding is essentially the same, we just have to mark the oracle tape and the query and answer state. By taking the lexicographic ordering on these strings, we get an ordering of the Turing machines.

We construct B iteratively. In the nth iteration, we add at most one string x_n of length n. B_n denotes the set that we have constructed in the first n iterations. We also have a set F of forbidden strings that shall not appear in B. This set is also constructed iteratively, F_n denotes the set that we have constructed in the first n iterations so far.

The nth iteration looks as follows. We simulate the nth Turing machine M_n with oracle B_{n-1} (with respect to the ordering defined above) on $\#^n$ for exactly 2^n steps. $M_n^{B_{n-1}}$ can query at most 2^{n-1} different strings, since for each query, we need one step to get the result and at least one step to write an oracle string. (In fact there are even fewer strings that the machine can query, since the oracle strings have to get longer and longer and the oracle tape is erased every time.) Let S_n be the set of strings that $M_n^{B_{n-1}}$ queries. Set $F_n = F_{n-1} \cup S_n$. If $M_n^{B_{n-1}}$ halts and rejects, we set $B_n = B_{n-1} \cup \{x_n\}$ for an arbitrary string in $\{0,1\}^n \setminus F_n$. Otherwise, we set $B_n = B_{n-1}$.

We have to check that the string x_n always exists. Since a Turing machine can query at most 2^{i-1} strings in 2^i steps,

$$|\bigcup_{1\le i\le n} S_i| \le \sum_{i=1}^{n} 2^{i-1} < 2^n.$$

Hence x_n exists.

Next we show that $\text{tally}(B)$ is not accepted by a deterministic Turing machine with running time 2^n. Assume that M_n is such a machine. M_n^B behaves like $M_n^{B_{n-1}}$ on input $\#^n$, since all strings that are queried by M_n are in S_n. These strings are not in B by construction. Hence B_{n-1} and B do not contain any of the strings queried by M_n.

The assumption that M_n^B decides $\#^n \in \text{tally}(B)$ correctly within 2^n steps yields a contradiction: If $\#^n \notin \text{tally}(B)$, then $M_n^{B_{n-1}}$ would reject $\#^n$, but then B would contain a string of length n, namely x_n, and $\#^n \in \text{tally}(B)$, a contradiction. If $\#^n \in \text{tally}(B)$, then $M_n^{B_{n-1}}$ would accept, but then $B_n = B_{n-1}$ and B would not contain a string of length n which implies that $\#^n \notin \text{tally}(B)$, a contradiction.

Thus it remains to show that $B \in \mathsf{EXP}$.[1] To decide whether a given x of length n is in B we enumerate the first n Turing machines and simulate them on $\#^i$. In this way, we can compute B_n. Once we have B_n, we can decide whether $x \in B$, since in later iterations, only longer strings are added

[1] This statement is only needed to say something about the complexity of B. We want to find an "easy" oracle.

to B. The ith simulation needs time $2^{O(i)}$. One oracle query is simulated by a look up in the table for B_{i-1}. Thus the total running time is $2^{O(n)}$. ■

Note that the oracle B achieves the largest possible gap between determinism and nondeterminism: tally(B) is in linear time recognizable by a nondeterministic Turing machine with oracle B but cannot be accepted by a 2^n time bounded deterministic Turing machine.

Exercise 11.1 *Prove that* $\text{tally}(B) \in \mathsf{DTime}^B(2^{O(n)})$.

Theorem 11.3 *There is a language* $B \in \mathsf{EXP}$ *such that* $\mathsf{P}^B \neq \mathsf{NP}^B$.

Proof. Let B be the language from Lemma 11.2. We have $\text{tally}(B) \in \mathsf{NP}^B$ and $\text{tally}(B) \notin \mathsf{DTime}^B(2^n)$. But then $\text{tally}(B)$ cannot be in P^B. ■

Pitfalls

It is not clear how to speed up oracle computations by arbitrary constant factors, since this could mean asking two queries in one step.

11.2 PH and PSPACE

In this section we will show the first part of a clever construction that separates PH from PSPACE with respect to some oracle. Before we can do so, we first have to define what PH^B means! We can set $\mathsf{PH}^B = \bigcup_d (\Sigma_d^{\mathsf{P}})^B$, so it remains to define $(\Sigma_d^{\mathsf{P}})^B$. Let M be a polynomial time deterministic oracle Turing machine such that M with oracle B computes some $(d+1)$-ary relation $R^B(x, y_1, \ldots, y_d)$. Then the language L defined by

$$x \in L \iff \exists^P y_1 \forall^P y_2 \ldots Q y_d : R^B(x, y_1, \ldots, y_d) = 1$$

is in $(\Sigma_d^{\mathsf{P}})^B$.

The separation result for PH and PSPACE depends on a lower bound for the size of Boolean circuits with unbounded fanin. A Boolean circuit with unbounded fanin is a Boolean circuit that can have inner nodes of arbitrary fanin. These nodes are labeled with $\vee$ or $\wedge$ and compute the Boolean conjunction or disjunction of the inputs. The size of the circuit is the number of nodes in it plus an additional $d-2$ for every node of fanin $d > 2$. The lower bound that is needed for the separation result was shown using the polynomial method. Recall that PARITY is the set of all $x \in \{0,1\}^*$ that have an odd number of ones, i.e., the parity of x is 1.

Theorem 11.4 (Furst, Saxe & Sipser) *If for all constants $d, c \geq 1$, there does not exist any family of unbounded fanin Boolean circuits of depth d and size $2^{\mathrm{O}(\log^c n)}$ computing* PARITY*, then there is an oracle B such that $\mathsf{PH}^B \neq \mathsf{PSPACE}^B$.*

Proof overview: For a language $A \subseteq \{0,1\}^*$, let $A^{=n} := A \cap \{0,1\}^n$ be the subset of all strings of length n. For any set A let

$$\pi(A,n) = \begin{cases} 1 & \text{if } |A^{=n}| \text{ is odd} \\ 0 & \text{otherwise} \end{cases}$$

In other words $\pi(A,n)$ is the parity of the number of words of length n in A. Let $P(A) = \{0^n \mid \pi(A,n) = 1\}$. We now view the characteristic vector a of all strings of length n of A as an input of length 2^n. Then $\texttt{PARITY}(a) = 1$ iff $0^n \in P(A)$. In this way, the oracle becomes the input of the circuit.

We will have the following mapping of concepts:

characteristic vectors of A	$\mapsto$	inputs of circuit
relation R	$\mapsto$	small subcircuits
quantifiers	$\mapsto$	unbounded fanin $\vee$ and $\wedge$

Proof. Let $P(A)$ be defined as above. We first show that there is a linear space bounded Turing machine M that given an oracle A, recognizes $P(A)$:

Input: $x \in \{0,1\}^*$

1. If $x \notin \{0\}^*$, then reject.
2. $c := 0$.
3. Enumerate all strings y in $\{0,1\}^{|x|}$, check whether $y \in A$, and if yes, then $c := c + 1 \mod 2$.
4. If $c = 1$ accept, else reject.

In particular, $P(A) \in \mathsf{PSPACE}^A$.

Let R be a $(d+1)$-ary relation. We call R good for length n if for all oracles A,

$$0^n \in P(A) \iff \exists^P y_1 \forall^P y_2 \ldots Q y_d : R^A(0^n, y_1, \ldots, y_d).$$

We now show that if a relation R is good for almost all lengths n (here used in the sense of "for all but finitely many") and can be computed in polynomial time, then we get a family of circuits that violates the assumption of the theorem.

Let N be a polynomial time bounded deterministic oracle Turing machine. We call N a fool if the following holds: For all n_0, for all $X \subseteq \{0,1\}^{\leq n_0}$, and

for infinitely many $n > n_0$, there exists a $Y \subseteq \{0,1\}^{>n_0}$ such that $N^{X \cup Y}$ errs on computing $R^{X \cup Y}(0^n, y_1, \ldots, y_d)$.

If N is not a fool, then we call N wise. For a wise N, there exists an n_0 and an X as above such that, for all but a finite number of $n > n_0$, $N^{X \cup Y}$ computes $R^{X \cup Y}(0^n, y_1, \ldots, y_d)$ correctly for all Y.

We can assume that N does not query strings of length $\ell \neq n$ and $\ell > n_0$. If N attempts to query such strings, then we can give it any value back since N has to answer correctly for all Y. Furthermore, there are only a finite number of strings of length at most n_0. Thus, instead of asking the oracle $X \cup Y$, the set X can be hard-wired into N. We will now show that wise N do not exist.

For fixed n, we can view the characteristic vector a of all strings of length n of A as an additional input of R; thus R becomes a function $r_n(y_1, \ldots, y_d, a) := R^A(0^n, y_1, \ldots, y_d)$.

Since N is $p(n)$ time bounded for some polynomial p, it can query the oracle at most $p(n)$ times for fixed n, $y_1, \ldots, y_d$. We now fix $y_1, \ldots, y_d$, and get a function $r_{n,y_1,\ldots,y_d}(a) := r_n(y_1, \ldots, y_d, a)$. The words that N queries may however depend on the answers to previous queries. We now simulate N on $0^n, y_1, \ldots, y_d$. Whenever N queries the oracle, we simulate it with possible answer 0 and with possible answer 1. We get a binary tree T of depth $p(n)$. The nodes are labeled with the corresponding strings queried and the edges are labeled with 0 or 1, the answer of the oracle to the query.

From this tree T, we get a formula $B_{n,y_1,\ldots,y_d}$ of depth two and size $\mathrm{O}(p(n) \cdot 2^{p(n)})$ that computes $r_{n,y_1,\ldots,y_d}$:

$$r_{n,y_1,\ldots,y_d}(a) = \bigvee_{\text{accepting paths } P \text{ of } T} a_{i_1^P}^{e_1^P} \wedge \cdots \wedge {}^{e_{p(n)}^P}_{i_{p(n)}^P}$$

In this disjunction, $i_1^P, \ldots, i_{p(n)}^P$ are the indices of the strings queried on the path P (i.e., the labels of the nodes along P) and $e_1^P, \ldots, e_{p(n)}^P$ are the answers on the path P (i.e., the labels of the edges along P). Above, we use the convention $x^1 = x$ and $x^0 = \neg x$ for a Boolean variable x.

$B_{n,y_1,\ldots,y_d}$ can be constructed by just simulating N for each possible outcome of the queries in space $p(n)$ (given $y_1, \ldots, y_d$).

Thus we get

$$0^n \in P(A) \iff \bigvee_{y_1} \bigwedge_{y_2} \bigvee_{y_3} \ldots B_{n,y_1,\ldots,y_d}(a).$$

But the righthand side is a circuit C_n of depth $d+2$ for PARITY. Its size is $\mathrm{O}(2^{dq(n)+p(n)})$ where q is a bound on the length of $y_1, \ldots, y_d$. It can be constructed in space $\mathrm{O}(dq(n) + p(n))$ which is logarithmic in the size. The number of inputs is $m := 2^n$. With respect to the number of inputs, the size of C_n is $2^{\mathrm{O}(\log^c m)}$ for some constant c.

Thus, if N is wise, then we can construct a family C_n that contradicts the assumption of the theorem. We only have circuits of input lengths 2^n but we can get a circuit for parity of any length ℓ out of it by rounding to the nearest power of 2 and then setting an appropriate number of inputs to 0.

Finally, we construct the oracle B separating PH^B from PSPACE^B by diagonalization. Every polynomial time Turing machine N_i that computes a potential relation R_i, is a fool. We now construct B inductively. For every candidate R_i, we choose a number $n_i > n_{i-1}$ that is greater than all previously chosen numbers such that, for all $X \in \{0,1\}^{\leq n_{i-1}}$, there exists a $Y \in \{0,1\}^{>n_{i-1}}$ such that N_i errs in computing $R^{X \cup Y}$ for length n_i. In particular, for $\bigcup_{\ell \leq n_{i-1}} B_i$, there exist Y_i and n_i with that property. We set $B_i = Y_i \cap \{0,1\}^{n_i}$ and $B = \bigcup_i B_i$. N_i with oracle B computes a relation that is not good for n_i, since we can always assume that N_i on input 0^n only queries strings of length n. Thus no R_i with oracle B is good for length n_i and hence, $P(B) \notin \mathsf{PH}^B$. ∎

Remark 11.5 *We can reduce the depth of C_n to $d+1$ by choosing B to be in CNF or DNF. We can then bring it down even to d by using the distributive laws and the fact that the gates at the bottom have fanin only $p(n)$.*

Excursus: Random oracle hypothesis

Not long after Theorem 11.3 was proven, Bennett and Gill showed that with respect to a random oracle A, i.e., every word x is in A with probability $1/2$, $\mathsf{P}^A \neq \mathsf{NP}^A$ with probability 1. This observation led to the random oracle hypothesis: Whenever a separation result is true with probability 1 with respect to a random oracle, then the unrelativized result should also be true. It was finally shown that $\mathsf{IP}^A \neq \mathsf{PSPACE}^A$ with probability 1 with respect to a random oracle. We will prove that $\mathsf{IP} = \mathsf{PSPACE}$ later on (and of course define IP).

---

12 Probabilistic computations

12.1 Randomized complexity classes

Probabilistic Turing machines have an additional *random tape*. On this tape, the Turing machine gets an one-sided infinite $\{0,1\}$ string y. The random tape is read-only and one-way.

Right at the moment, we are considering the random string y as an additional input. The name random string is justified by the following definition: A probabilistic Turing machine accepts an input x with *acceptance probability* at least p if $\Pr[M(x,y)=1] \geq p$. Here the probability is taken over all choices of y. We define the *rejection probability* in the same way. The running time $t(n)$ of a probabilistic Turing machine M is the maximum number of steps that M performs on any input of length n and and any random string y. Note that if $t(n)$ is bounded, then we can consider y to be a finite string of length at most $t(n)$. The maximum number of random bits a Turing machine reads on any input x of length n and random string y is called the amount of randomness used by the machine.

We define $\mathsf{RTime}(t(n))$ to be the class of all languages L such that there is a Turing machine M with running time $t(n)$ and for all $x \in L$, M accepts x with probability at least $1/2$ and for all $x \notin L$, M rejects L with probability 1. Such an M is said to have a *one-sided error*. If M in fact accepts each $x \in L$ with probability $\geq 1-\epsilon \geq 1/2$, then we say that the error probability of M is bounded by ϵ.

The class $\mathsf{BPTime}(t(n))$ is defined in the same manner, but we allow the Turing machine M to err in two ways. We require that for all $x \in L$, M accepts x with probability at least $2/3$ and for all $x \notin L$, M rejects with probability at least $2/3$ (that is, accepts with probability at most $1/3$). Such an error is called a *two-sided error*. If M actually accepts all $x \in L$ with probability $\geq 1-\epsilon$ and accepts each $x \notin L$ with probability $\leq \epsilon$, then we say that the error probability is bounded by ϵ.

Definition 12.1 *1.* $\mathsf{RP} = \bigcup_{i\in\mathbb{N}} \mathsf{RTime}(n^i)$,

2. $\mathsf{BPP} = \bigcup_{i\in\mathbb{N}} \mathsf{BPTime}(n^i)$,

3. $\mathsf{ZPP} = \mathsf{RP} \cap \mathsf{co\text{-}RP}$.

The name ZPP stands for *zero error probabilistic polynomial time*. It is justified by the following statement.

Exercise 12.1 *A language L is in* ZPP *if and only if L is accepted by a probabilistic Turing machine with error probability zero and expected polynomial running time. Here the expectation is taken over all possible random strings on the random tape.*

For robust classes (such as RP and BPP) the choice of the constants $1/2$ and $2/3$ in the definitions of RTime and BPTime is fairly arbitrary, since both classes allow *probability amplification.*

Lemma 12.2 *Let M be a Turing machine for some language $L \in$ RP that runs in time $t(n)$, uses $r(n)$ random bits, and has error probability ϵ. For any $k \in \mathbb{N}$, there is a Turing machine M' for L that runs in time $O(kt(n))$, uses $kr(n)$ random bits, and has error probability ϵ^k.*

Proof. M' works as follows:

Input: $x \in \{0,1\}^*$

1. M' simulates M k times, each time using new random bits.
2. M' accepts, if in at least one of the simulations, M accepts. Otherwise, M' rejects.

The bounds on the time and randomness are obvious. If $x \notin L$, then M' also rejects, since M does not err on x. If $x \in L$, then with probability at most ϵ, M rejects x. Since M' performs k independent trials, the probability that M' rejects x is at most ϵ^k. ∎

Lemma 12.3 *Let M be a Turing machine for some language $L \in$ BPP that runs in time $t(n)$, uses $r(n)$ random bits, and has error probability $\epsilon < 1/2$. For any $k \in \mathbb{N}$, there is a Turing machine M' for L that runs in time $O(kt(n))$, uses $kr(n)$ random bits, and has error probability $2^{-c_\epsilon k}$ for some constant c_ϵ that solely depends on ϵ.*

Proof. M' works as follows:

Input: $x \in \{0,1\}^*$

1. M'simulates M k times, each time with fresh random bits.
2. M' accepts, if in at least half of the simulations (rounded up), M accepts. Otherwise, M' rejects.

Let μ be the expected number of times that a simulated run of M accepts. If $x \in L$, then $\mu \geq (1-\epsilon)k$. The probability that less than half of the simulated runs of M accept is $< e^{-\frac{(1-\epsilon)\delta^2}{2}k}$ with $\delta = 1 - \frac{1}{2(1-\epsilon)}$ by the Chernoff bound (see below). The case $x \notin L$ is treated similarly. In both cases, the error probability is bounded by 2^{ck} for some constant c only depending on ϵ. ∎

Remark 12.4 *In both lemmas, k can also be a function in n, as long as k is computable in time $\mathrm{O}(k(n)t(n))$. (All reasonable functions k are.)*

In the proof above, we used the so-called *Chernoff bound*. A proof of it can be found in most books on probability theory.

Lemma 12.5 (Chernoff bound) *Let $X_1, \ldots, X_m$ be independent 0–1 valued random variables and let $X = X_1 + \cdots + X_m$. Let $\mu = \mathrm{E}(X)$. Then for any $\delta > 0$,*

$$\Pr[X > (1+\delta)\mu] < \left(\frac{e^\delta}{(1+\delta)^{1+\delta}}\right)^\mu \quad \textit{and} \quad \Pr[X < (1-\delta)\mu] < e^{-\frac{\mu\delta^2}{2}}.$$

We have

$$\mathsf{P} \subseteq \mathsf{ZPP} \subseteq \begin{matrix}\mathsf{RP} \\ \mathsf{co\text{-}RP}\end{matrix} \subseteq \mathsf{BPP}. \tag{12.1}$$

The latter two inclusions follow by amplifying the acceptance probability once.

Exercise 12.2 *Let* PP *be the class of all languages L such that there is a polynomial time probabilistic Turing machine M that accepts all $x \in L$ with probability $\geq 1/2$ and accepts all $x \notin L$ with probability $< 1/2$. Show that* NP $\subseteq$ PP.

12.2 Relation to other classes

We start with comparing RP and BPP with non-randomized complexity classes. The results in this section are the basic ones, further results will be treated in the lecture "Pseudorandomness and Derandomization" in the next semester.

Theorem 12.6 BPP $\subseteq$ PSPACE.

Proof. Let M be polynomial time bounded Turing machine with error probability bounded by $1/3$ for some $L \in \mathsf{BPP}$. Assume that M reads at most $r(n)$ random bits on inputs of length n. Turing machine M' simulates M as follows:

Input: $x \in \{0,1\}^*$

1. M' systematically lists all bit strings of length $r(n)$.
2. M' simulates M with the current string as random string.
3. M' counts how often M accepts and rejects.

4. If the number of accepting computations exceeds the number of rejecting computations, M' accepts. Otherwise, M' rejects.

Since M is polynomial time, $r(n)$ is bounded by a polynomial. Hence M' uses only polynomial space. ■

Corollary 12.7 $\mathsf{BPP} \subseteq \mathsf{EXP}$.

Theorem 12.8 $\mathsf{RP} \subseteq \mathsf{NP}$.

Proof. Let M be a polynomial time probabilistic Turing machine with error probability bounded by $1/2$ for some $L \in \mathsf{RP}$. We convert M into a nondeterministic machine M' as follows: Whenever M would read a bit from the random tape, M' nondeterministically branches to the two states that M would enter after reading zero or one, respectively.

If M does not accept x, then there is no random string such that M on input x reaches an accepting configuration. Thus there is no accepting path in the computation tree of M' either.

On the other hand, if M accepts x, then M reaches an accepting configuration on at least half of the random strings. Thus at least half of the paths in the computation tree of M' are accepting ones. In particular, there is at least one accepting path. Hence M' accepts x. ■

NP **and** RP

NP: *one* proof/witness of membership
RP: *many* proofs/witnesses of membership

Next we turn to the relation between BPP and circuits.

Theorem 12.9 (Adleman) $\mathsf{BPP} \subseteq \mathsf{P/poly}$.

Proof. Let $L \in \mathsf{BPP}$. By Lemma 12.3, there is a polynomial time bounded probabilistic Turing machine with error probability $< 2^{-n}$ that accepts L. There are 2^n possible input strings of length n. Since for each string x of length n, the error probability of M is $< 2^{-n}$, M can err on x only for a fraction of all possible random strings that is smaller than 2^{-n}. Thus there must be one random string that is good for all inputs of length n. We take this string as an advice string for the inputs of length n. By Lemma 10.2, $L \in \mathsf{P/poly}$. ■

How do we find this good random string? If we amplify the error probability even further, say to 2^{-2n}, then almost all, namely a fraction of $1 - 2^{-n}$ random strings are good. Thus picking the advice at random is a good strategy. (This, however, requires randomness!)

12.3 Further exercises

Exercise 12.3 *Show the following: If* $\mathtt{SAT} \in \mathsf{BPP}$*, then* $\mathtt{SAT} \in \mathsf{RP}$*. (Hint: downward self-reducibility).*

The answer to the following question is not known.

Open Problem 12.10 *Prove or disprove:* $\mathsf{RP} = \mathsf{P}$ *implies* $\mathsf{BPP} = \mathsf{P}$.

13 The BP-operator

BPP extends P by adding randomness to the computation but all other properties of P stay the same. In this section, we introduce a general mechanism of adding randomness to a complexity class.

Definition 13.1 *Let* C *be a class of languages. The class* BP-C *is the class of all languages A such that there is a $B \in$ C, a polynomial p, and constants $\alpha \in (0,1)$ and $\beta > 0$ such that for all inputs x:*

$$x \in A \Longrightarrow \Pr_{y \in \{0,1\}^{p(|x|)}}[\langle x, y\rangle \in B] \geq \alpha + \beta/2,$$
$$x \notin A \Longrightarrow \Pr_{y \in \{0,1\}^{p(|x|)}}[\langle x, y\rangle \in B] \leq \alpha - \beta/2,$$

α is called the *threshold value* and β is the *probability gap*. Note the similarity to the $\exists$- and $\forall$-operators.

Exercise 13.1 *Prove that* BPP = BP-P.

13.1 Probability amplification

Definition 13.2 *Let* C *be a class of languages.* BP-C allows probability amplification *if for every $A \in$ BP-C and every polynomial q, there is a language $B \in$ C and a polynomial p such that for all inputs x:*

$$x \in A \Longrightarrow \Pr_{y \in \{0,1\}^{p(|x|)}}[\langle x, y\rangle \in B] \geq 1 - 2^{-q(|x|)}$$
$$x \notin A \Longrightarrow \Pr_{y \in \{0,1\}^{p(|x|)}}[\langle x, y\rangle \in B] \leq 2^{-q(|x|)}$$

We would like to be able to perform probability amplification. For BPP, we ran the Turing machine several times and took a majority vote. Now the key point is that the resulting machine is again a probabilistic polynomial time machine.

A sufficient condition for a class C to allow probability amplification is that for all $L \in$ C, $\mathsf{P}^L \in$ C, i.e., C is closed under Turing reductions. But this is a too strong restriction, since it is not clear whether for instances NP is closed under Turing reductions. However, a weaker condition also suffices.

Definition 13.3 *1. A* positive polynomial time Turing reduction *from a language A to a language B is a polynomial time Turing machine R*

such that $A = L(R^B)$ and R has the following monotonicity property: if $D \subseteq D'$ then $L(R^D) \subseteq L(R^{D'})$ for all languages D, D'. We write $A \leq_{\mathrm{P}}^{\mathrm{T}+} B$ in this case.

2. *Let B be a language and C be a class of languages. We define $\mathsf{Pos}(B) = \{A \mid A \leq_{\mathrm{P}}^{\mathrm{T}+} B\}$ and $\mathsf{Pos}(\mathsf{C}) = \{A \mid A \leq_{\mathrm{P}}^{\mathrm{T}+} B$ for some $B \in \mathsf{C}\}$.*

Exercise 13.2 *Prove the following:* $\mathsf{Pos}(\mathsf{NP}) = \mathsf{NP}$.

Lemma 13.4 *Let C be closed under positive polynomial time Turing reductions. Then $\mathsf{BP}\text{-}\mathsf{C}$ allows probability amplification.*

Proof. Let $A \in \mathsf{BP}\text{-}\mathsf{C}$ and let $D \in \mathsf{C}$, $\alpha \in (0,1)$, and $\beta > 0$ such that the conditions in Definition 13.1 holds. Let q be a polynomial.

Let $k = k(|x|)$ be some integer to be chosen later. The language B now consists of all $\langle x, y_1, \ldots, y_k\rangle$ such that for more than αk indices i, $\langle x, y_i\rangle \in D$. (I.e., the y_κ serve as fresh random strings for k independent trials. We then take a majority vote.)

Since there is a positive Turing reduction from B to D (if $D \subseteq D'$, then $\langle x, y_i\rangle \in D$ of course implies $\langle x, y_i\rangle \in D'$), $B \in \mathsf{C}$. If we now choose $k = \mathrm{O}(q(|x|))$, then the error probability goes down to $2^{-q(|x|)}$ (as already seen for BPP). ∎

Exercise 13.3 *Show the following: If $\mathsf{BP}\text{-}\mathsf{C}$ allows probability amplification, then $\mathsf{BP}\text{-}\mathsf{BP}\text{-}\mathsf{C} = \mathsf{BP}\text{-}\mathsf{C}$.*

13.2 Operator swapping

Lemma 13.5 *If C is closed under Pos, then*

1. $\exists\,\mathsf{BP}\text{-}\mathsf{C} \subseteq \mathsf{BP}\text{-}\exists\mathsf{C}$,
2. $\forall\,\mathsf{BP}\text{-}\mathsf{C} \subseteq \mathsf{BP}\text{-}\forall\mathsf{C}$,

Proof. Let $A \in \exists\,\mathsf{BP}\text{-}\mathsf{C}$. By assumption, there is a language $B \in \mathsf{BP}\text{-}\mathsf{C}$ such that for all x,

$$x \in A \iff \exists^P b : \langle x, b\rangle \in B. \tag{13.1}$$

In particular, $x \in A$ depends only on $B^{\leq p(|x|)}$, the strings in B of length at most $p(|x|)$ for some polynomial p.

Since $\mathsf{BP}\text{-}\mathsf{C}$ allows probability amplification, for every polynomial q, there is a language $D \in \mathsf{C}$ and a polynomial r such that for all inputs y:

$$y \in B \Longrightarrow \Pr_{z \in \{0,1\}^{r(|y|)}}[\langle y, z\rangle \in D] \geq 1 - 2^{-q(|y|)},$$

$$y \notin B \Longrightarrow \Pr_{z \in \{0,1\}^{r(|y|)}}[\langle y, z\rangle \in D] \leq 2^{-q(|y|)}.$$

Set $q(n) = 2n + 3$. It follows that for all n:

$$\begin{aligned}\Pr_z[\text{for all } y \text{ with } |y| \le p(n)\text{: } y \in B \iff \langle y, z\rangle \in D] &\ge 1 - \sum_{\nu=0}^{p(n)} 2^{-q(\nu)} \cdot 2^{\nu} \\ &\ge 1 - 1/4 \\ &= 3/4,\end{aligned}$$

by the union bound. For a string z, let $Y(z) := \{y \mid \langle y, z\rangle \in D\}$. Thus for all n

$$\Pr_z[B^{\le p(n)} = Y(z)^{\le p(n)}] \ge 3/4.$$

This means that for a random z, $Y(z)^{\le p(n)}$ behaves like $B^{\le p(n)}$ with high probability. Finally set

$$\begin{aligned}E &= \{\langle x, z\rangle \mid \exists^P b\ \langle x, b\rangle \in Y(z)^{\le p(|x|)})\} \\ &= \{\langle x, z\rangle \mid \exists^P b\ \langle\langle x, b\rangle, z\rangle \in D \wedge |\langle x, b\rangle| \le p(|x|)\}.\end{aligned}$$

Since $D \in \mathsf{C}$, $E \in \exists\mathsf{C}$, as C is closed under Pos. (To test the predicate, we first check whether $|\langle x, b\rangle| \le p(|x|)$, and if it is, we then query D.) This means that in (13.1), we can replace the righthand side by

"for a fraction of $\ge 3/4$ of all z, $\langle x, z\rangle \in E$".

Thus $A \in \mathsf{BP}\text{-}\exists\mathsf{C}$.

The case of $\forall$ is shown in exactly the same way. ■

Exercise 13.4 *Show that if* $\mathsf{NP} \subseteq \mathsf{BPP}$*, then* $\Sigma_2^{\mathsf{P}} \subseteq \mathsf{BPP}$ *(and even* $\mathsf{PH} \subseteq \mathsf{BPP}$*).*

13.3 BPP and the polynomial hierarchy

For two strings $u, v \in \{0,1\}^m$, $u \oplus v$ denotes the string that is obtained by taking the bitwise XOR.

Lemma 13.6 (Lautemann) *Let* $n > \log m$*. Let* $S \subseteq \{0,1\}^m$ *with* $|S| \ge (1 - 2^{-n})2^m$*.*

1. *There are* $u_1, \ldots, u_m$ *such that for all* v*,* $u_i \oplus v \in S$ *for some* $1 \le i \le m$*.*
2. *For all* $u_1, \ldots, u_m$ *there is a* v *such that* $u_i \oplus v \in S$ *for all* $1 \le i \le m$*.*

Proof.

1. We have

$$\begin{aligned}
&\Pr_{u_1,\ldots,u_m\in\{0,1\}^m}[\exists v : u_1 \oplus v, \ldots, u_m \oplus v \notin S] \\
&\leq \sum_{v\in\{0,1\}^m} \Pr_{u_1,\ldots,u_m\in\{0,1\}^m}[u_1 \oplus v, \ldots, u_m \oplus v \notin S] \\
&= \sum_{v\in\{0,1\}^m} \prod_{i=1}^{m} \Pr_{u_i\in\{0,1\}^m}[u_i \oplus v \notin S] \\
&\leq 2^m \cdot (2^{-n})^m \\
&< 1,
\end{aligned}$$

since $u_i \oplus v$ distributed uniformly in $\{0,1\}^m$ and all the u_i's are drawn independently. Since the probability that a v with the desired properties does not exist is < 1, there must be a v that fulfills the assertions of the first claim.

2. Fix $u_1, \ldots, u_m$. We have

$$\begin{aligned}
\Pr_{v\in\{0,1\}^m}[\exists i : u_i \oplus v \notin S] &\leq \sum_{i=1}^{m} \Pr_{v\in\{0,1\}^m}[u_i \oplus v \notin S] \\
&\leq m \cdot 2^{-n} \\
&< 1.
\end{aligned}$$

Thus a v exists such that for all i, $u_i \oplus v \in S$. Since $u_1, \ldots, u_m$ were arbitrary, we are done.

■

Lemma 13.7 *Let* C *be a complexity class such that* $\mathsf{Pos}(\mathsf{C}) = \mathsf{C}$. *Then*

1. $\mathsf{BP}\text{-}\mathsf{C} \subseteq \exists\forall\mathsf{C}$ *and*
2. $\mathsf{BP}\text{-}\mathsf{C} \subseteq \forall\exists\mathsf{C}$.

Proof. Let A be a language in $\mathsf{BP}\text{-}\mathsf{C}$. Since C is closed under Pos, we can do probability amplification: There is a language $B \in \mathsf{C}$ and some polynomial p such that for all x,

$$\begin{aligned}
x \in A &\Longrightarrow \Pr_{y\in\{0,1\}^{p(|x|)}}[\langle x, y\rangle \in B] \geq 1 - 2^{-n}, \\
x \notin A &\Longrightarrow \Pr_{y\in\{0,1\}^{p(|x|)}}[\langle x, y\rangle \in B] \leq 2^{-n}.
\end{aligned}$$

Let $T_x = \{y \mid \langle x, y\rangle \in B\}$. If $x \in A$, then $|T_x| \geq (1 - 2^{-n})2^{p(|x|)}$. In this case, the first statement of Lemma 13.6 is true. If $x \notin A$, then $|\bar{T}_x| \geq (1-2^{-n})2^{p(|x|)}$.

Thus the second statement of Lemma 13.6 is true for $\bar{T}_x$. But this is the negation of the first statement for T_x. Hence

$$x \in A \iff \exists^P u_1, \ldots, u_{p(|x|)} \forall^P v : u_1 \oplus v \in T_x \vee \cdots \vee u_{p(|x|)} \oplus v \in T_x.$$

The relation on the righthand side clearly is in $\mathsf{Pos}(\mathsf{C})$, and hence, $A \in \exists\forall\mathsf{C}$.

In the same way, we get $A \in \forall\exists\mathsf{C}$, since if $x \in A$, then also the second statement of Lemma 13.6 is true for T_x and if $x \notin A$, then the first statement is true for $\bar{T}_x$. ■

Corollary 13.8 (Sipser) $\mathsf{BPP} \subseteq \Sigma_2^\mathsf{P} \cap \Pi_2^\mathsf{P}$.

Corollary 13.9 *If* $\mathsf{P} = \mathsf{NP}$, *then* $\mathsf{P} = \mathsf{BPP}$.